Minecra

src_resources

Table of contents

1. MinecraftForge 文档（中文翻译）

这里是Minecraft模组API——MinecraftForge的官方文档。

该文档 仅 针对Forge编纂，而不是一个**Java**教程。

如果你愿意对文档做出贡献，请阅读向文档做出贡献。

2. 向这篇文档做出贡献

你可以通过在MinecraftForge的官方GitHub仓库提交PR来提交对这篇文档的贡献；若要对翻译文档提出修改，请访问中文翻译GitHub仓库。

本文档的宗旨是明理达意。因此在提交贡献时，请解释它的工作原理，并将其拆分为合理的部分。 我们在其他地方有一个wiki，可以展示更全面的代码示例（未指明具体地点——译者注）。

我们的受众是任何一位想要了解如何通过Forge来构筑模组的人。

请不要试图将这篇文档转变为一篇关于Java开发的教程——后者意在教导人们一个Java类是如何工作的，以及一些Java的其他基本架构。

2.1 格式指南

> **重要**
>
> 请使用两个空格符来进行缩进，而不是制表符。

标题应该以标准的标题样式对首字母进行大写或小写。例如，

- Guide For Contributing to This Documentation （中文：本文档贡献指南）
- Building and Testing Your Mod （中文：构建并测试你的模组）

从本质上讲，除了不重要的单词外，所有单词都要大写。

拼写、语法和句法应遵循美式英语。此外，优先考虑使用用空格分割开的短语而不是缩写（例如“are not”而不是“aren't”）。

请使用等号和短划线，而不要使用“#”和“##”。对于h3及以下字体大小，“###”等也可以。此文件的源文件包含一个等号和短划线的示例。等号下划线创建h1文本，短划线下划线创建h2文本。

当引用代码块片段之外的字段和方法时，它们应该使用 `#` 分隔符（例如 `ClassName#methodName` ）。内部类应该使用 `$` 分隔符（例如 `ClassName$InnerClassName` ）。

JSON代码块片段应该使用 `js` 语法高亮显示。

所有链接都应在页面底部指定其URL。任何本文档的内部链接都应该通过其相对路径引用对应的页面。

警告（用 `!!! <type>` 表示）必须按照该文档所示进行设置；否则，它们最终可能会渲染不正确。

3. 入门

3.1 Forge入门

如果你之前从未制作过一个Forge模组，本节将提供设置Forge开发环境所需的最少信息。其余的文档是关于从这里开始的内容。

3.1.1 先决条件

- 安装Java 17开发包（JDK）和64位JVM。Forge推荐并官方支持Eclipse Temurin。

> **警告**
>
> 确保你正在使用64位的JVM。一种检查方式是在终端中运行 `java -version` 。使用32位的JVM会导致在使用ForgeGradle的过程中出现问题。

- 熟练使用一款集成开发环境（IDE）。
  - 建议使用一款集成了Gradle功能的IDE。

3.1.2 从零开始模组开发

1. 从Forge文件站下载Mod开发包（MDK）。点击“Mdk”，等待一段时间之后点击右上角的“Skip”按钮。如果可能的话，推荐下载最新版本的Forge。
2. 解压所下载的MDK到一个空文件夹中。它会成为你的模组的目录，且现在应该已包含一些gradle文件和一个含有example模组的 `src` 子目录。

> **注意**
>
> 许多文件可以在不同的模组中重复使用。这些文件是：
>
> - `gradle` 子目录
> - `build.gradle`
> - `gradlew`
> - `gradlew.bat`
> - `settings.gradle`
>
> `src` 子目录不需要跨工作区进行复制；但是，如果稍后创建java（`src/main/java`）和resource（`src/main/resources`），则可能需要刷新Gradle项目。

3. 打开你选择的IDE：
   - Forge只明确支持在Eclipse和IntelliJ IDEA上进行开发，但还有其他针对Visual Studio代码的运行配置。无论如何，从Apache NetBeans到Vim/Emacs的任何开发环境都可被使用。
   - Eclipse和IntelliJ IDEA的Gradle集成，都是已默认安装和启用的，将在导入或打开时处理其余的初始工作区设置。这包括从Mojang、MinecraftForge等下载必要的软件包。如果你使用Visual Studio，则需要安装“Gradle for Java”插件。
   - Gradle将需要被调用来重新评估项目中对其相关文件的几乎所有更改（如 `build.gradle` 、 `settings.gradle` 等等）。有些IDE带有“刷新”按钮来完成此操作；然而，它也可以通过在终端上运行 `gradlew` 来完成。

4. 为你选择的IDE生成运行配置:
   - **Eclipse**: 运行 `genEclipseRuns` 任务。
   - **IntelliJ IDEA**: 运行 `genIntellijRuns` 任务。如果发生了"module not specified"错误，请将 `ideaModule` 属性设置为你的'main'模块（通常为 `${project.name}.main`）。
   - **Visual Studio Code**: 运行 `getVSCodeRuns` 任务。
   - **Other IDEs**: 你可以通过 `gradle run*` 来直接运行这些配置（如 `runClient`、`runServer`、`runData`、`runGameTestServer`）。这对于已提供支持的IDE同样有效。

3.1.3 自定义你的模组信息

编辑 `build.gradle` 文件以自定义你的模组的构建方式（如文件名称、artifact版本等等）。

> **重要**
>
> 除非你知道你在做什么，否则不要编辑 `settings.gradle`。该文件指定ForgeGradle所上传的仓库。

建议的 `build.gradle` 自定义项目

Mod Id替换

将包括mods.toml和主mod文件在内的所有出现的examplemo替换为你的模组的mod id。这还包括通过设置 `base.archivesName`（通常设置为你的mod id）来更改你构建的文件的名称。

```
// 在某个build.gradle文件中
base.archivesName = 'mymod'
```

> **注意**
>
> Forge MDK目前使用 `archivesBaseName` 来设置artifact名称，而不是 `base.archivesName`。我们建议使用 `base.archivesName` 作为替代，因为 `archivesBaseName` 由于在Gradle 9中被移除而被废弃，而ForgeGradle的未来版本将支持此功能。
>
> 你仍然可以通过以下设置使用 `archivesBaseName`:
>
> ```
> // 在某个build.gradle文件中
> base.archivesName = 'mymod'
> ```

Group Id

`group` 属性应该设置为你的顶级程序包，其应为你拥有的域名或你的电子邮件地址:

类型	值	顶级程序包
域名	example.com	`com.example`
子域名	example.github.io	`io.github.example`
电子邮箱地址	example@gmail.com	`com.gmail.example`

```
// 在某个build.gradle文件中
group = 'com.example'
```

java源文件（`src/main/java`）中的包现在也应该符合这种结构，更深层的包表示mod id：

```
com
- example (在group属性中所指定的顶级程序包)
  - mymod (mod id)
    - MyMod.java (重命名后的ExampleMod.java)
```

版本

将 `version` 属性设置为你的模组的当前版本。我们推荐采用Maven版本号命名格式。

```
// 在某个build.gradle文件中
version = '1.19.4-1.0.0.0'
```

额外配置

额外配置可在ForgeGradle文档中找到。

3.1.4 构建并测试你的模组

1. 要构建你的模组，请运行 `gradlew build` 。这将在 `build/libs` 输出一个默认名为 `[archivesBaseName]-[version].jar` 的文件。这个文件可以被放在已安装了Forge的Minecraft的 `mods` 文件夹中，也可以被分发。
2. 要在测试环境中运行你的模组，你既可以使用已生成的运行配置，也可以运行功能类似的Gradle任务（例如 `gradlew runClient` ）。这将使用任何所指定的源码集从run文件夹中启动Minecraft。默认的MDK包括 `main` 源码集，因此任何在 `src/main/java` 中编写的源代码都会被应用。
3. 如果你想要运行dedicated服务端，无论是通过运行配置，还是通过 `gradlew runServer` ，服务端都会立刻宕机。你需要通过编辑run文件夹中的 `eula.txt` 文件同意Minecraft EULA。一旦同意后，服务器就会加载，之后就可以通过直连 `localhost` 进行访问了。

注意

在服务端环境测试你的模组是必要的。这包括只针对客户端的模组，因为在加载到服务端后它们不应该做任何事。

3.2 模组文件

模组文件负责确定哪些文件会被打包到你模组的JAR文件中，在“Mods”菜单中显示哪些信息，以及你的模组如何被加载到游戏中。

3.2.1 mods.toml

`mods.toml` 定义你的一个或多个模组的元数据。它也包含一些附加信息，这些信息将在Mods菜单中被展示，并决定你的模组如何被加载进游戏。

该文件采用Tom’s Obvious Minimal Language（简称TOML）格式。这个文件必须保存在你所使用的源码集的resource目录中的 `META-INF` 文件夹下（例如对于 `main` 源码集，其路径为 `src/main/resources/META-INF/mods.toml` ）。 `mods.toml` 文件看起来长这样：

```
modLoader="javafml"
loaderVersion="[45,)"

license="All Rights Reserved"
issueTrackerURL="https://github.com/MinecraftForge/MinecraftForge/issues"
showAsResourcePack=false

[[mods]]
  modId="examplemod"
  version="1.0.0.0"
  displayName="Example Mod"
  updateJSONURL="https://files.minecraftforge.net/net/minecraftforge/forge/promotions_slim.json"
  displayURL="https://minecraftforge.net"
  logoFile="logo.png"
  credits="I'd like to thank my mother and father."
  authors="Author"
  description='''
  Lets you craft dirt into diamonds. This is a traditional mod that has existed for eons. It is ancient. The holy Notch created it.
  '''
  displayTest="MATCH_VERSION"

[[dependencies.examplemod]]
  modId="forge"
  mandatory=true
  versionRange="[45,)"
  ordering="NONE"
  side="BOTH"

[[dependencies.examplemod]]
  modId="minecraft"
  mandatory=true
  versionRange="[1.19.4]"
  ordering="NONE"
  side="BOTH"
```

`mods.toml` 被分为三个部分：非模组特定属性，与模组文件相关联；模组特定属性，对每个模组都有单独的小节；以及依赖配置，对每个模组依赖都有单独的小节。下面将解释与 `mods.toml` 文件相关的各个属性，其中 `required` 表示必须指定一个值，否则将引发异常。

非模组特定属性

非模组特定属性是与JAR文件本身相关的属性，指明如何加载模组和任何附加的全局元数据。

属性	类型	缺省值	描述	样例
`modLoader`	string	必需	模组所使用的语言加载器。可用于支持额外的语言结构，如为主文件定义的Kotlin对象，或确定入口点的不同方法，如接口或方法。Forge提供Java加载器 `"javafml"` 和低/无代码加载器 `"lowcodefml"` 。	`"javafml"`
`loaderVersion`	string	必需	可接受的语言加载器版本范围，以Maven版本范围表示。对于 `javafml` 和 `lowcodefml` ，其版本是Forge版本的主版本号。	`"[45,)"`
`license`	string	必需	该JAR文件中的模组所遵循的许可证。建议将其设置为你正在使用的SPDX标识符和/或许可证的链接。你可以访问 https://choosealicense.com/ 以帮助选取你想使用的许可证。	`"MIT"`
`showAsResourcePack`	boolean	`false`	当为 `true` 时，模组的资源会以一个单独的资源包的形式在“资源包”菜单中展示，而不是与“模组资源”包融为一体。	`true`
`services`	array	`[]`	表示你的模组所使用的一系列服务的数组。这是从Forge的Java平台模块系统实现中为模组创建的模块的一部分。	`["net.minecraftforge.forgespi.language.IModLanguageProvi`
`properties`	table	`{}`	替换属性表。 `StringSubstitutor` 使用它将 `${file.<key>}` 替换为相应的值。该功能目前仅用	由 `${file.example}` 引用的 `{ "example" = "1.2.3" }`

属性	类型	缺省值	描述	样例
			于替换模组特定属性中的 `version` 。	
`issueTrackerURL`	string	无	指向报告与追踪模组问题的地点的URL。	`"https://forums.minecraftforge.net/"`

重要

`services` 属性在功能上等效于在指定在模块中的 `uses` 指令，该指令允许加载给定类型的服务。

模组特定属性

模组特定属性通过 `[[mods]]` 头与指定的模组绑定。其本质是一个表格数组（Array of Tables）；直到下一个头之前的所有键/值对都会被关联到那个模组。

```
# examplemod1的属性
[[mods]]
modId = "examplemod1"

# examplemod2的属性
```

```
[[mods]]
modId = "examplemod2"
```

属性	类型	缺省值	描述	样例
`modId`	string	必需	代表这个模组的唯一标识符。该标识符必须匹配 `^[a-z][a-z0-9_]{1,63}$` （一个长度在[2,64]闭区间内的字符串；以小写字母开头；由小写字母、数字或下划线组成）。	`"examplemod"`
`namespace`	string	`modId` 的值	该模组的一个重载命名空间。该命名空间必须匹配 `^[a-z][a-z0-9_.-]{1,63}$` （一个长度在[2,64]闭区间内的字符串；以小写字母开头；由小写字母、数字、下划线、点或短横线组成）。目前无作用。	`"example"`
`version`	string	`"1"`	该模组的版本，最好符合Maven版本号命名格式。当设置为 `${file.jarVersion}` 时，它将被替换为JAR清单文件中 `Implementation-Version` 属性的值（在开发环境下默认显示为 `0.0NONE` ）。	`"1.19.4-1.0.0.0"`
`displayName`	string	`modId` 的值	该模组的更具可读性的名字。用于将模组展示到屏幕上时（如模组列表、模组不匹配）。	`"Example Mod"`
`description`	string	`"MISSING DESCRIPTION"`	在模组列表中展示的该模组的描述。建议使用一个多行文字字符串。	`"This is an example."`
`logoFile`	string	无	在模组列表中展示的该模组的logo图像文件的名称和扩展名。该logo必须位于JAR文件的根目录或直接位于源码集的根目录。	`"example_logo.png"`
`logoBlur`	boolean	`true`	决定使用 `GL_LINEAR*` （true）或 `GL_NEAREST*` （false）渲染 `logoFile` 。	`false`
`updateJSONURL`	string	无	被更新检查器用来检查你所使用的模组是否为最新版本的指向一个JSON文件的URL。	`"https://files.minecraftforge.net/net/mi`

属性	类型	缺省值	描述	样例
`features`	table	`{}`	参见 ‘features‘。	`{ java_version = "17" }`
`modproperties`	table	`{}`	与本模组相关联的一个键/值对表。目前尚未被Forge使用，但主要被模组使用。	`{ example = "value" }`
`modUrl`	string	无	指向本模组下载界面的URL。目前无作用。	`"https://files.minecraftforge.net/"`
`credits`	string	无	在模组列表中展示的致谢声明。	`"The person over here and there."`
`authors`	string	无	在模组列表中展示的本模组的作者。	`"Example Person"`
`displayURL`	string	无	在模组列表中展示的本模组的展示页面（项目主页）。	`"https://minecraftforge.net/"`
`displayTest`	string	`"MATCH_VERSION"`	参见 ‘sides‘。	`"NONE"`

功能

功能系统允许模组在加载系统时要求某些设置、软件或硬件可用。当某个功能不满足时，模组加载将失败，并将要求通知给用户。目前，Forge提供以下功能：

功能	描述	样例
`java_version`	可支持的Java版本范围，以Maven版本范围表示。该范围须能够支持Minecraft所使用的Java版本。	`"[17,)"`

依赖配置

模组可以指定它们的依赖项，这些依赖项在加载模组之前由Forge检查。这些配置是使用表格数组（Array of Tables）`[[dependencies.<modid>]]` 创建的，其中 `modid` 是所依赖的模组的标识符。

属性	类型	缺省值	描述	样例
`modId`	string	必需	被添加为依赖的模组的标识符。	`"example_library"`
`mandatory`	boolean	必需	当依赖未满足时游戏是否崩溃。	`true`
`versionRange`	string	`""`	可接受的语言加载器版本范围，以Maven版本范围表示。空字符串表示匹配所有版本。	`"[1, 2)"`
`ordering`	string	`"NONE"`	定义本模组是否必须在所依赖的模组之前（`"BEFORE"`）或之后（`"AFTER"`）加载。`"NONE"` 表示不规定顺序。	`"AFTER"`
`side`	string	`"BOTH"`	所依赖模组必须位于的端位：`"CLIENT"`、`"SERVER"` 或 `"BOTH"`。	`"CLIENT"`
`referralUrl`	string	无	指向依赖下载界面的URL。目前无作用。	`"https://library.example.com/"`

警告

两个模组的 `ordering` 可能会因循环依赖而造成崩溃：例如模组A必须在模组B之前（`"BEFORE"`）加载，而模组B也必须在模组A之前（`"BEFORE"`）加载。

3.2.2 模组入口点

现在我们已经填写了 `mods.toml`，我们需要提供一个对模组进行编程的入口点。入口点本质上是执行模组的起点。入口点本身由 `mods.toml` 中使用的语言加载器决定。

`javafml` 和 `@Mod`

`javafml` 是Forge为Java编程语言提供的语言加载器。入口点是通过使用带有 `@Mod` 注释的公共类来定义的。`@Mod` 的值必须包含 `mods.toml` 中指定的一个Mod id。从那里，所有初始化逻辑（例如注册事件、添加 `DeferredRegister`）都可以在类的构造函数中写明。模组总线可以从 `FMLJavaModLoadingContext` 获得。

```
@Mod("examplemod") // 必须匹配mods.toml
public class Example {

  public Example() {
    // 此处初始化逻辑
    var modBus = FMLJavaModLoadingContext.get().getModEventBus();

    // ...
  }
}
```

lowcodefml

`lowcodefml` 是一种语言加载器，用于将数据包和资源包作为模组形式分发，而无需代码形式的入口点。它被指定为 `lowcodefml` 而不是 `nocodefml`，用于将来可能需要的最少量代码的小添加。

3.3 规划你的模组结构

结构分明的模组有利于维护和做出贡献，并提供对底层代码库的更清晰理解。下面列举了由Java、Minecraft和Forge提出的一些建议。

注意

你不必遵循以下建议；你可以以任何你认为合适的方式规划你的模组。然而，我们仍强烈建议这样做。

3.3.1 程序包

在规划你的模组时，选择一个独特的、顶级的程序包结构。许多程序员会对不同的类、接口等使用相同的名称。Java允许类具有相同的名称，只要它们位于不同的包中。因此，如果两个类具有相同的名称和相同的包，则只有一个会被加载，这很可能导致游戏崩溃。

```
a.jar
  - com.example.ExampleClass
b.jar
  - com.example.ExampleClass // 这个类不会被正常加载
```

当涉及到加载模块时，这一点更为重要。如果在不同的模块中有两个同名包下的类文件，这将导致模组加载器在启动时崩溃，因为模组模块会被导出到游戏和其他模组中。

```
module A
  - package X
    - class I
    - class J
module B
  - package X // 此包将导致模组加载器崩溃，因为已经有一个模块将包X导出
    - class R
    - class S
    - class T
```

正因如此，你的顶级程序包应该是你自己的东西：域名、电子邮件地址、网站的子域等。它甚至可以是你的名字或用户名，只要你能保证它在预期目标中是唯一可识别的。

类型	值	顶级程序包
域名	example.com	`com.example`
子域名	example.github.io	`io.github.example`
电子邮箱地址	example@gmail.com	`com.gmail.example`

下一个级别的包应该是你的mod id（例如 `com.example.examplemod`，其中 `examplemod` 是mod id）。这将保证，除非你有两个id相同的模组（这种情况永远不会发生），否则你的包在加载时不会出现任何问题。

你可以在Oracle的教程页面上找到一些其他命名约定。

子包的组织

除了顶级包以外，强烈建议将你的模组的类拆分为不同的子包。关于如何做到这一点，主要有两种方法：

- 按功能分组：将具有共同目的的类归入同一个子包。例如，方块相关的类可被置于 `block` 或 `blocks` 子包下，实体相关的类可被置于 `entity` 或 `entities` 子包下等等。Mojang就在使用这种结构，单词用的是单数形式（ `block` 、 `entity` ）。
- 按逻辑分组：将具有共同逻辑的类归入同一个子包。例如，如果你正在创建一种新配方，你可以将它的方块、菜单、物品等等都放在 `feature.crafting_table` 子包下。

客户端、服务端和数据相关的子包

通常，仅用于给定端位或运行时的代码都应该在单独的子包中与其他类隔离。例如，与数据生成相关的代码应该放在 `data` 子包中，而仅与dedicated服务器相关的代码应该在 `server` 子包中。

然而，强烈建议在 `client` 子包中隔离仅限客户端的代码。这是因为dedicated服务器不应有任何权限访问Minecraft中仅限客户端的包。因此，拥有一个专用的包将提供一个不错的健全性检查，以保证你的模组中的代码没有越过端位的行为。

3.3.2 类的命名规则

一个普适的类命名方案可以让你更容易地读懂类的目的或查找某个特定的类。

类的名称通常以其类型作为后缀，例如：

- 一个叫作 `PowerRing` 的 `Item` -> `PowerRingItem` 。
- 一个叫作 `NotDirt` 的 `Block` -> `NotDirtBlock` 。
- 为 `Oven` 设计的一个菜单 -> `OvenMenu` 。

注意

Mojang通常对除实体以外的所有类命名时都遵循类似的结构。而实体只用它们的名字来表示（例如 `Pig` 、 `Zombie` 等）。

3.3.3 选择仅用一个方法而非多个

执行特定任务的方法有很多：注册对象、监听事件等。通常建议使用单一方法来完成给定的任务以保持一致。这在改善了代码格式的同时，也避免了可能发生的任何奇怪的交互或冗余（例如，你的事件监听器执行了两次）。

3.4 版本号

在一般项目中，语义式的版本号（格式为 `MAJOR.MINOR.PATCH` ）被经常使用。然而，在长期性地修改的情况下，使用格式 `MCVERSION-MAJORMOD.MAJORAPI.MINOR.PATCH` 可能更有利于将模组的创世性的修改与API变更性的修改区分开来。

重要

Forge使用Maven版本范围来比较版本字符串，这与Semantic Versioning 2.0.0规范不完全兼容，例如“prerelease”标签。

3.4.1 样例

以下是在不同情形下能递增各种变量的示例列表。

- `MCVERSION`
- 始终与该模组所适用的Minecraft版本相匹配。
- `MAJORMOD`
- 移除物品、方块、方块实体等。
- 改变或移除之前存在的机制。
- 升级到新的Minecraft版本。
- `MAJORAPI`
- 更改枚举的顺序或变量。
- 更改方法的返回类型。
- 一并移除公共方法。
- `MINOR`
- 添加物品、方块、方块实体等。
- 添加新机制。
- 废弃公共方法。（这不是一次 `MAJORAPI` 递增，因为它并未改变API。）
- `PATCH`
- Bug修复。

当递增任何变量时，所有更小级别的变量都应重置为 `0` 。例如，如果 `MINOR` 递增， `PATCH` 将变为 `0` 。如果 `MAJORMOD` 递增，则所有其他变量将变为 `0` 。

项目初始阶段

如果你正处于模组的初始开发阶段（在任何正式发布之前）， `MAJORMOD` 和 `MAJORAPI` 应始终为 `0` 。只有 `MINOR` 和 `PATCH` 应该在每次构建你的模组时更新。一旦你构建了一个官方版本（大多数情况下应使用稳定的API），你应该将 `MAJORMOD` 增加到版本 `1.0.0.0` 。有关任何进一步的开发阶段，请参阅本文档的预发布和候选发布部分。

多个**Minecraft**版本

如果模组升级到新版本的Minecraft，而旧版本将只会得到bug修复，则 `PATCH` 变量应根据升级前的版本进行更新。如果模组针对旧版本和新版本的Minecraft都仍在积极开发中，建议将该版本附加到所有两个Minecraft版本号之后。例如，如果模组由于Minecraft版本的更改而升级到 `3.0.0.0` 版本，那么旧版本的模组也应该更新到 `3.0.0.0` 。又例如，旧版本将变成 `1.7.10-3.0.0.0` 版本，而新版本将变成 `1.8-3.0.0.0` 版本。如果在为新的Minecraft版本构建时模组本身并没有任何更改，那么除了Minecraft版本之外的所有变量都应该保持不变。

最终发布

当放弃对某个Minecraft版本的支持时，针对该版本的最后一个模组构建版本应该有 `-final` 后缀。这意味着模组对于所表示的 `MCVERSION` 将不再支持，玩家应该升级到模组所支持的新版本的Minecraft，以继续接收更新和bug修复。

预发布

（本指南不使用 `-pre`，因为在撰写本文时，它不是 `-beta` 的有效别名。）请注意，已经发布的版本和首次发布之前的版本不能进入预发布；变量（主要是 `MINOR`，但 `MAJORAPI` 和 `MAJORMOD` 也可以预发布）应该在添加 `-beta` 后缀之前进行相应的更新。首次发布之前的版本只是在建版本。

候选发布

候选发布在实际版本更替之前充当预发布。这些版本应该附加 `-rcX`，其中 `X` 是候选版本的数量，理论上，只有在修复bug时才应该增加。已经发布的版本无法接收候选版本；在添加 `-rc` 后缀之前，应该相应地更新变量（主要是 `MINOR`，但 `MAJORAPI` 和 `MAJORMOD` 也可以预发布）。当作为稳定构建版本发布候选版本时，它既可以与上一个候选版本完全相同，也可以有更多的bug修复。

4. 核心概念

4.1 注册表

注册是获取模组的对象（如物品、方块、音效等）并使其为游戏所知的过程。注册东西很重要，因为如果没有注册，游戏将根本不知道这些对象，这将导致无法解释的行为和崩溃。

游戏中的大多数注册相关事项都由Forge注册表处理。注册表是一个与为键分配值的Map的行为类似的对象。Forge使用带有 `ResourceLocation` 键的注册表来注册对象。这允许 `ResourceLocation` 充当对象的“注册表名称”。

每种类型的可注册对象都有自己的注册表。要查看由Forge封装的所有注册表，请参阅 `ForgeRegistries` 类。注册表中的所有注册表名称必须是唯一的。但是，不同注册表中的名称不会发生冲突。例如，有一个 `Block` 注册表和一个 `Item` 注册表。一个方块和一个物品可以用相同的名称 `example:thing` 注册而不冲突；但是，如果两个不同的方块（或物品）以相同的名称被注册，则第二个对象将覆盖第一个对象。

4.1.1 注册的方式

有两种正确的方式来注册对象： `DeferredRegister` 类和 `RegisterEvent` 生命周期事件。

DeferredRegister

`DeferredRegister` 是注册对象的推荐方式。它包容静态初始化的使用与便利，同时也避免与之相关的问题。它只需维护一系列的Supplier，并在 `RegisterEvent` 期间注册这些Supplier所提供的对象。（Supplier是Java 8加入的新语法。——译者注）

以下是一个模组注册一个自定义方块的案例：

```
private static final DeferredRegister<Block> BLOCKS = DeferredRegister.create(ForgeRegistries.BLOCKS, MODID);

public static final RegistryObject<Block> ROCK_BLOCK = BLOCKS.register("rock", () -> new Block(BlockBehaviour.Properties.of(Material

public ExampleMod() {
  BLOCKS.register(FMLJavaModLoadingContext.get().getModEventBus());
}
```

RegisterEvent

`RegisterEvent` 是注册对象的第二种方式。在模组构造函数之后和加载configs之前，该事件会为每个注册表激发。对象通过调用 `#register` 并传入注册表键、注册表对象的名称和对象本身而得以注册。还有一个额外的 `#register` 重载，它接收一个已使用的助手来注册具有给定名称的对象。建议使用此方法以避免不必要的对象创建。

案例如下：（事件处理器已被注册到模组事件总线）

```
@SubscribeEvent
public void register(RegisterEvent event) {
  event.register(ForgeRegistries.Keys.BLOCKS,
    helper -> {
      helper.register(new ResourceLocation(MODID, "example_block_1"), new Block(...));
      helper.register(new ResourceLocation(MODID, "example_block_2"), new Block(...));
      helper.register(new ResourceLocation(MODID, "example_block_3"), new Block(...));
      // ...
```

```
    }
  );
}
```

未被**Forge**封装的注册表

并非所有的注册表都由Forge封装。这些可以是静态注册表，如 `LootItemConditionType` ，使用起来是安全的。还有动态注册表，如 `ConfiguredFeature` 和其他一些世界生成注册表，它们通常以JSON表示。 `DeferredRegister#create` 有一个重载，允许模组开发者指定原版注册表所创建的 `RegistryObject` 的注册表键。注册表方法和模组事件总线的附加与其他 `DeferredRegister` 相同。

重要

动态注册表对象只能通过数据文件（如JSON）被注册。它们不能在代码中被注册。

```
private static final DeferredRegister<LootItemConditionType> REGISTER = DeferredRegister.create(Registries.LOOT_CONDITION_TYPE, "exa

public static final RegistryObject<LootItemConditionType> EXAMPLE_LOOT_ITEM_CONDITION_TYPE = REGISTER.register("example_loot_item_cc
```

注意

有些类无法自行注册。相反， `*Type` 类被注册，并在前者的构造函数中被使用。例如， `BlockEntity` 具有 `BlockEntityType` ， `Entity` 具有 `EntityType` 。这些 `*Type` 类是工厂，它们只是根据需要创建包含类型。

这些工厂是通过使用它们的 `*Type$Builder` 类创建的。例如： （ `REGISTER` 指的是 `DeferredRegister<BlockEntityType>` ）

```
public static final RegistryObject<BlockEntityType<ExampleBlockEntity>> EXAMPLE_BLOCK_ENTITY = REGISTER.register(
  "example_block_entity", () -> BlockEntityType.Builder.of(ExampleBlockEntity::new, EXAMPLE_BLOCK.get()).build(null)
);
```

4.1.2 引用已注册的对象

已注册的对象在创建和注册时不应存储在字段中。每当为相应的注册表触发 `RegisterEvent` 时，它们应总是新创建并注册的。这是为了允许在未来版本的Forge中动态加载和卸载模组。

已注册的对象必须始终通过 `RegistryObject` 或带有 `@ObjectHolder` 的字段引用。

使用**RegistryObjects**

一旦注册对象可用，就可以使用 `RegistryObjects` 检索对这些对象的引用。 `DeferredRegister` 使用它们来返回对已注册对象的引用。在为其注册表触发 `RegisterEvent` 后，它们的引用以及带有 `@ObjectHolder` 注释的字段都将被更新。

要获取 `RegistryObject` ，请使用可注册对象的 `IForgeRegistry` 和一个 `ResourceLocation` 调用 `RegistryObject#create` 。亦可使用自定义注册表，方式是向其提供注册表名称。请将 `RegistryObject` 存储在一个 `public static final` 字段中，并在需要该已注册对象时调用 `#get` 。

使用 `RegistryObject` 的一个案例：

```
public static final RegistryObject<Item> BOW = RegistryObject.create(new ResourceLocation("minecraft:bow"), ForgeRegistries.ITEMS);

// 假设'neomagicae:mana_type'是一个合法的注册表，且'neomagicae:coffeinum'是该注册表中一个合法的对象
public static final RegistryObject<ManaType> COFFEINUM = RegistryObject.create(new ResourceLocation("neomagicae", "coffeinum"), new
```

使用**@ObjectHolder**

通过使用 `@ObjectHolder` 注释类或字段，并提供足够的信息来构造 `ResourceLocation` 以标识特定注册表中的特定对象，可以将注册表中的已注册对象注入 `public static` 字段。

使用 `@ObjectHolder` 的规则如下：

- 若类被使用 `@ObjectHolder` 注释，则如果未明确定义，其值将是该类中所有字段的默认命名空间
- 若类被使用 `@Mod` 注释，则如果未明确定义，modid将是其中所有已注释字段的默认命名空间
- 若符合下列条件，该类中的一个字段将会被考虑注入：
- 其至少包含修饰符 `public static`；
- 该字段被 `@ObjectHolder` 注释，并且：
  - name值已被显式指明；并且
  - registry name值已被显式指明
- 如果某个字段没有相应的注册表（*registry name*）或名称（*name*），则会引发编译时异常。
- 如果最终的 `ResourceLocation` 不完整或无效（路径中存在无效字符），则会引发异常。
- 如果没有发生其他错误或异常，则该字段将被注入
- 如果以上所有规则都不适用，则不会采取任何操作（并且日志可能会输出一条信息）

被 `@ObjectHolder` 注释的字段会在 `RegisterEvent` 为其注册表激发之后注入其值，与 `RegistryObjects` 的引用的更新同时发生。

> **注意**
>
> 如果要注入对象时该对象不存在于注册表中，那么日志会记录一条调试信息，并且不会注入任何值。

由于这些规则相当复杂，案例如下：

```
class Holder {
  @ObjectHolder(registryName = "minecraft:enchantment", value = "minecraft:flame")
  public static final Enchantment flame = null;     // 注释存在。[public static]是必需的。[final]是可选的。
                                                    // Registry name已被显式指明："minecraft:enchantment"
                                                    // Resource location已被显式指明："minecraft:flame"
                                                    // 将注入：[Enchantment]注册表中的"minecraft:flame"

  public static final Biome ice_flat = null;        // 该字段无注释。
                                                    // 因此，该字段被忽略。

  @ObjectHolder("minecraft:creeper")
  public static Entity creeper = null;              // 注释存在。[public static]是必需的。
                                                    // 该字段未指明注册表。
                                                    // 因此，其将引发编译时异常。

  @ObjectHolder(registryName = "potion")
  public static final Potion levitation = null;     // 注释存在。[public static]是必需的。[final]是可选的。
                                                    // Registry name已被显式指明："minecraft:potion"
                                                    // Resource location未在该字段中指明
                                                    // 因此，其将引发编译时异常。
}
```

4.1.3 创建自定义的Forge注册表

自定义注册表通常只是一个简单的键值映射。这是一种常见的风格；然而，它强制对存在的注册表进行严格的依赖。它还要求任何需要在端位之间同步的数据都必须手动完成。自定义Forge注册表为创建软依赖项提供了一个简单的替代方案，同时提供了更好的管理手段和端位之间的自动同步（除非另有说明）。由于这些对象也使用Forge注册表，注册也以同样的方式标准化。

自定义Forge注册表是在 `RegistryBuilder` 的帮助下通过 `NewRegistryEvent` 或 `DeferredRegister` 创建的。 `RegistryBuilder` 类接受多种参数（例如注册表的名称、id范围以及注册表上发生的不同事件的各种回调）。 `NewRegistryEvent` 完成激发后，新的注册表将被注册到 `RegistryManager` 。

任何新创建的注册表都应该使用其关联的注册方法来注册关联的对象。

使用**NewRegistryEvent**

使用 `NewRegistryEvent` 时，用 `RegistryBuilder` 调用 `#create` 将返回一个用Supplier包装的注册表。 `NewRegistryEvent` 在模组事件总线处理完毕后，这个Supplier注册表就可以访问了。在 `NewRegistryEvent` 被处理完毕之前试图从Supplier获取该自定义注册表将得到 `null` 值。

使用**DeferredRegister**

`DeferredRegister` 方法又是上述事件的另一个包装。一旦使用 `#create` 重载在常量字段中创建了 `DeferredRegister` （该重载接受注册表名称和mod id），就可以通过 `DeferredRegistry#makeRegistry` 构建注册表。该方法接受了由Supplier提供的包含任何其他配置的 `RegistryBuilder` 。默认情况下，该方法已调用 `#setName` 。由于此方法可以在任何时候返回，因此会返回由Supplier提供的 `IForgeRegistry` 版本。在激发NewRegistryEvent之前试图从Supplier获取自定义注册表将得到 `null` 值。

> **重要**
>
> 在通过 `#register` 将 `DeferredRegister` 添加到模组事件总线之前，必须调用 `DeferredRegister#makeRegistry` 。 `#makeRegistry` 也使用 `#register` 方法在 `NewRegistryEvent` 期间创建注册表。

4.1.4 处理缺失的注册表条目

在某些情况下，每当更新模组或删除模组（更可能的情况）时，某些注册表对象将不复存在。可以通过第三个注册表事件指定操作来处理丢失的映射： `MissingMappingsEvent` 。在该事件中，既可以通过给定注册表项和mod id的 `#getMappings` 获取丢失映射的列表，也可以通过给定注册项的 `#getAllMappings` 获取所有映射。

> **重要**
>
> `MissingMappingsEvent` 在**Forge**事件总线上触发。

对于每个映射（ `Mapping` ），可以选择四种映射类型之一来处理丢失的条目：

操作	描述
IGNORE	忽略丢失的条目并丢弃映射。
WARN	在日志中生成警告。
FAIL	阻止世界加载。
REMAP	将条目重新映射到已注册的非null对象。

如果未指定任何操作，则默认操作为通过通知用户丢失的条目以及用户是否仍要加载世界。除了重新映射之外的所有操作都将防止任何其他注册表对象取代现有id，以防止相关条目被添加回游戏中。

4.2 Minecraft中的端位

为Minecraft开发模组时需要理解的一个非常重要的概念是两个端位：客户端和服务端。关于端位有很多常见的误解和错误，这可能会导致bug，而这些bug虽然可能不会破坏游戏，但是一定能够产生意想不到的、往往不可预测的影响。

4.2.1 不同种类的端位

当我们说“客户端”或“服务端”时，我们通常会对所谈论的游戏的哪个部分有相当直观的理解。毕竟，客户端是用户交互的对象，服务端是用户连接多人游戏的地方。很简单，对吧?

而事实是，即使有两个这样的术语，也可能存在一些歧义。在这里，我们消除了“客户端”和“服务端”的四个可能含义的歧义：

- 物理客户端 - 无论何时从启动器启动Minecraft，物理客户端都是运行的整个程序。在游戏的图形化、可交互的生命周期中运行的所有线程、进程和服务都是物理客户端的一部分。
- 物理服务端 - 通常被称为dedicated服务端， 物理服务端是在你启动任何类型的 `minecraft_server.jar` 时运行的整个程序，该程序不会显示可用于游玩的GUI。
- 逻辑服务端 - 逻辑服务端运行游戏逻辑：生物的生成，天气，物品栏、生命值、AI的更新以及其他所有游戏机制。逻辑服务端存在于物理服务端中，但它也可以与逻辑客户端一起在物理客户端中运行，作为一个单机世界。逻辑服务端始终在名为 `Server Thread` 的线程中运行。
- 逻辑客户端 - 逻辑客户端接受玩家的输入并将其转发到逻辑服务端。此外，它还从逻辑服务端接收信息，并以图形方式呈现给玩家。逻辑客户端在 `Render Thread` 中运行，但通常会派生出几个其他线程来处理音频和方块渲染批处理等事务。

在MinecraftForge代码库中，物理端由一个名为 `Dist` 的枚举表示，而逻辑端则由一个名为 `LogicalSide` 的枚举表示。

4.2.2 进行特定端位的操作

`Level#isClientSide`

这种boolean检查将是你最常用的检查端位的方法。在 `Level` 对象上查询此字段将建立该Level所属的逻辑端。也就是说，如果此字段为 `true` ，则该Level当前正在逻辑客户端上运行。如果该字段为 `false` ，则表示该Level正在逻辑服务端上运行。因此，物理服务端在该字段中总是包含 `false` ，但我们不能假设 `false` 意味着物理服务端，因为该字段对于物理客户端（换句话说，单机世界）内的逻辑服务端也可能是 `false` 。

当你需要确定是否应该运行游戏逻辑和其他机制时，请使用这种检查方式。例如，如果你想在玩家每次点击你的方块时伤害他们，或者让你的机器将泥土处理成钻石，你只有在确保 `#isClientSide` 为 `false` 后才能这样做。在最好的情况下，将游戏逻辑应用于逻辑客户端可能会导致去同步（幽灵实体、去同步状态等），在最坏的情况下会导致崩溃。

这种检查应该成为习惯。你很少需要除 `DistExecutitor` 以外的其他方式来确定端位和调整行为。

`DistExecutor`

考虑到客户端和服务端的模组都使用同一个“通用”的jar，以及将物理端分离为两个jar，我们想到了一个重要的问题：我们该如何使用只存在于某一个物理端的代码？ `net.minecraft.client` 下的所有代码仅存在于物理客户端上。如果你编写的任何类以任何方式引用了上述包下的类型名称，那么当在不存在这些类型名称的环境中加载相应的类时，它们将导致游戏崩溃。初学者的一个非常常见的错误是在他们的方块或方块实体类中调用 `Minecraft.getInstance().<doStuff>()` ，一旦加载这些类，就会导致任何物理服务端崩溃。

我们如何解决这个问题？幸运的是，FML有一个 `DistExecutor` ，它提供了各种方法来在不同的物理端运行不同的方法，或者只在某一物理端运行单个方法。

注意

对FML基于物理端进行检查的理解尤为重要。单机世界（包含逻辑服务端+物理客户端的逻辑客户端）将始终使用 `Dist.CLIENT`！

`DistExecutor` 的工作原理是接收所提供的执行方法的Supplier，通过利用JVM指令 `invokedynamic` 有效地防止类加载。被执行的方法应该是静态的并且在不同的类中。此外，如果这个静态方法没有参数，则应使用该方法的引用，而不是一个执行方法的Supplier。

`DistExecutor` 中有两个主要方法： `#runWhenOn` 和 `#callWhenOn` 。方法接受的参数为将被执行的方法和该方法应该运行的物理端，该方法（将被执行的方法）既可有返回值，也可无返回值。

这两种方法被进一步细分为 `#safe*` 和 `#unsafe*` 变体。安全（safe）和不安全（unsafe）这两种命名方式其实差强人意。主要区别在于，在开发环境中， `#safe*` 方法将验证所提供的执行方法是否是返回的对另一个类的方法引用的lambda，否则将抛出错误。在产品环境中， `#safe*` 和 `#unsafe*` 在功能上是相同的。

```
// 在一个客户端类中：ExampleClass
public static void unsafeRunMethodExample(Object param1, Object param2) {
  // ...
}

public static Object safeCallMethodExample() {
  // ...
}

// 在一个通用类中
DistExecutor.unsafeRunWhenOn(Dist.CLIENT, () -> ExampleClass.unsafeRunMethodExample(var1, var2));

DistExecutor.safeCallWhenOn(Dist.CLIENT, () -> ExampleClass::safeCallMethodExample);
```

警告

由于 `invokedynamic` 在Java 9+中的工作方式发生了变化， `DistExecutor` 方法的所有 `#safe*` 变体都会在开发环境中抛出封装在 `BootstrapMethodError` 中的原始异常。应该使用 `#unsafe*` 变体或对 `FMLEnvironment#dist` 的检查作为替代。

线程组

如果 `Thread.currentThread().getThreadGroup() == SidedThreadGroups.SERVER` 为true，则很可能当前线程位于逻辑服务端上。否则，它很可能在逻辑客户端上。当你无法访问 `Level` 对象以检查 `isClientSide` 时，这对于检索逻辑端非常有用。它通过查看当前运行的线程组来猜测你处于哪个逻辑端。因为这是一种猜测，所以只有在用尽其他选项时才应该使用这种方法。在几乎所有情况下，你应该优先检查 `Level#isClientSide` 。

`FMLEnvironment#dist` 和 `@OnlyIn`

`FMLEnvironment#dist` 表示当前你的代码正在运行的物理端。由于它是在启动时确定的，所以它不依赖于猜测来返回结果。然而，在这方面的用例并不是很多。

使用 `@OnlyIn(Dist)` 注释对方法或字段进行注释会向加载器表明，应该将相应的成员在非指定的物理端中从定义里完全剥离。通常，这些只有在浏览反编译的Minecraft代码时才能看到，暗示着Mojang混淆器删除了的方法。没有理由直接使用此注释。请改用 `DistExecutor` 或检查 `FMLEnvironment#dist` 。

4.2.3 常见错误

跨逻辑端访问

每当你想将信息从一个逻辑端发送到另一个逻辑端时，必须始终使用网络数据包。即便在单机场景中，将数据从逻辑服务端直接传输到逻辑客户端是非常诱人的。

实际上，这通常是通过静态字段无意中完成的。由于在单机场景中，逻辑客户端和逻辑服务端共享相同的JVM，因此向静态字段写入和从静态字段读取的线程都会导致各种竞争条件以及与线程相关的经典问题。

通过从逻辑服务端上运行或可以运行的公共代码访问仅物理客户端的类（如 `Minecraft` ），也可能会明确地犯下这个错误。对于在物理客户端中调试的初学者来说，这个错误很容易被忽略。代码会在那里工作，但它会立即在物理服务端上崩溃。

4.2.4 编写单端模组

在最近的版本中，Minecraft Forge从mods.toml中删除了一个“sidedness”属性。这意味着无论你的模组是加载在物理客户端还是物理服务端上，它们都可以工作。因此，对于单端模组，你通常会在 `DistExecutor#safeRunWhenOn` 或 `DistExecutor#unsafeRunWhen` 中注册事件处理程序，而不是直接调用模组构造函数中的相关注册方法。基本上，如果你的模组加载在错误的一端，它应该什么都不做，不监听任何事件，等等。单端模组本质上不应该注册方块、物品……因为它们也需要在另一端可用。

此外，如果你的模组是单端的，它通常不会禁止用户加入缺乏该模组的服务端。因此，你应该将mods.toml中的 `displayTest` 属性设置为任何必要的值。

```
[[mods]]
  # ...

  # MATCH_VERSION表示如果客户端和服务端上的版本不同，你的模组将导致红X。这是默认行为，如果你的模组有服务端和客户端元素，这就是你应该
  # IGNORE_SERVER_VERSION表示如果你的模组出现在服务端上但不在客户端上，它不会导致红X。如果你的模组是一个仅限服务端的模组，这就是你应
  # IGNORE_ALL_VERSION表示如果你的模组出现在客户端或服务端上，它不会导致红X。这是一个特殊情况，只有当你的模组没有服务端成分时才应该(
  # NONE表示没有在你的模组上设置显示检测。你需要自己完成此操作，有关详细信息，请参阅IExtensionPoint.DisplayTest。你可以使用此值定义(
  # 重要提示：这不是关于你的模组加载在哪个环境（客户端或dedicated服务端）上的说明。你的模组必然会加载（也许什么都不做！）。
  displayTest="IGNORE_ALL_VERSION" # 如果未指定任何内容，则MATCH_VERSION为默认值（#可选）
```

如果要使用自定义显示检测，则 `displayTest` 选项应设置为 `NONE` ，并且应注册 `IExtensionPoint$displayTest` 扩展：

```
//确保另一个网络端上缺失的模组不会导致客户端将服务端显示为不兼容
ModLoadingContext.get().registerExtensionPoint(IExtensionPoint.DisplayTest.class, () -> new IExtensionPoint.DisplayTest(() -> Networ
```

这告诉客户端它应该忽略服务端版本不存在，服务端不应该告诉客户端这个模组应该存在。因此，这个代码片段适用于仅客户端和服务端的模组。

4.3 事件

Forge使用事件总线以允许模组拦截来自各种原版和模组行为的事件。

例如：右键单击原版的木棍时，一个事件可被触发以用于执行操作。

用于大多数事件的主事件总线位于 `MinecraftForge#EVENT_BUS` 。在 `FMLJavaModLoadingContext#getModEventBus` 中还有另一个用于特定于模组事件的事件总线，你应该只在特定情况下使用它。关于该事件总线的更多信息可以在下面找到。

每个事件都在其中一条总线上触发：大多数事件在主要的Forge事件总线上触发，但也有一些在特定于模组的事件总线上触发。

事件处理器是某个已注册到事件总线的方法。

4.3.1 创建一个事件处理器

事件处理器方法只有一个参数，不返回结果。该方法可以是静态的，也可以是实例化的，具体取决于实现。

事件处理器可以使用 `IEventBus#addListener` 直接注册，或对于泛型事件（ `GenericEvent<T>` 的子类）使用 `IEventBus#addGenericListener` 直接注册。任一监听器注册方法接收表示方法引用的Consumer。泛型事件处理器还需要指定泛型的具体类型。事件处理器必须在模组主类的构造函数中注册。

```
// 在模组主类ExampleMod中

// 该事件位于模组事件总线上
private void modEventHandler(RegisterEvent event) {
    // Do things here
}

// 该事件位于Forge事件总线上
private static void forgeEventHandler(AttachCapabilitiesEvent<Entity> event) {
    // ...
}

// 在模组构造函数内
modEventBus.addListener(this::modEventHandler);
forgeEventBus.addGenericListener(Entity.class, ExampleMod::forgeEventHandler);
```

实例化的已注释的事件处理器

该事件处理器监听 `EntityItemPickupEvent` ，正如名称所述，每当 `Entity` 拾取一件物品时，该事件就会被发布到事件总线。

```
public class MyForgeEventHandler {
    @SubscribeEvent
    public void pickupItem(EntityItemPickupEvent event) {
        System.out.println("Item picked up!");
    }
}
```

要注册这个事件处理器，请使用 `MinecraftForge.EVENT_BUS.register(...)` 并向其传递事件处理器所在类的一个实例。如果要将此处理器注册到特定于模组的事件总线，则应使用 `FMLJavaModLoadingContext.get().getModEventBus().register(...)` 。

静态的已注释的事件处理器

事件处理器也可以是静态的。处理事件的方法仍然使用 `@SubscribeEvent` 进行注释。与实例化的事件处理器的唯一区别是它也被标记为 `static` 。要注册静态的事件处理器，传入类的实例是不行的。必须传入类本身。例如：

```
public class MyStaticForgeEventHandler {
    @SubscribeEvent
    public static void arrowNocked(ArrowNockEvent event) {
        System.out.println("Arrow nocked!");
    }
}
```

其必须像这样注册： `MinecraftForge.EVENT_BUS.register(MyStaticForgeEventHandler.class)` 。

自动注册静态的事件处理器

类可以使用 `@Mod$EventBusSubscriber` 进行注释。当 `@Mod` 类本身被构造时，这样的类会自动注册到 `MinecraftForge#EVENT_BUS` 。这实质上相当于在 `@Mod` 类的构造函数的末尾添加 `MinecraftForge.EVENT_BUS.register(AnnotatedClass.class);` 。

你可以向 `@Mod$EventBusSubscriber` 注释指明所要监听的总线。建议你也指定mod id，因为注释在处理的过程中可能无法确定它，以及你所注册的总线，因为它作为一个保障可以确保你所注册的是正确的总线。你还可以指定要加载此事件处理器的 `Dist` 或物理端。这可用于保证不在dedicated服务器上加载客户端特定的事件处理器。

下面是静态事件处理器监听 `RenderLevelStageEvent` 的示例，该处理器将仅在客户端上调用：

```
@Mod.EventBusSubscriber(modid = "mymod", bus = Bus.FORGE, value = Dist.CLIENT)
public class MyStaticClientOnlyEventHandler {
    @SubscribeEvent
    public static void drawLast(RenderLevelStageEvent event) {
        System.out.println("Drawing!");
    }
}
```

注意

这不会注册类的实例；它注册类本身（即事件处理方法必须是静态的）。

4.3.2 事件的取消

如果一个事件可以被取消，它将带有 `@Cancelable` 注释，并且方法 `Event#isCancelable()` 将返回 `true` 。可取消事件的取消状态可以通过调用 `Event#setCanceled(boolean canceled)` 来修改，其中传递布尔值 `true` 意为取消事件，传递布尔值 `false` 被解释为“不取消”事件。但是，如果无法取消事件（如 `Event#isCancelable()` 所定义），则无论传递的布尔值如何，都将抛出 `UnsupportedOperationException` ，因为不可取消事件事件的取消状态被认为是不可变的。

重要

并非所有事件都可以取消！试图取消不可取消的事件将导致抛出未经检查的 `UnsupportedOperationException` ，可能将导致游戏崩溃！在尝试取消某个事件之前，请始终使用 `Event#isCancelable()` 检查该事件是否可以取消！

4.3.3 事件的结果

某些事件具有 `Event$Result` 。结果可以是以下三种情况之一： `DENY` （停止事件）、 `DEFAULT` （使用默认行为）和 `ALLOW` （强制执行操作，而不管最初是否执行）。事件的结果可以通过调用 `#setResult` 并用一个 `Event$Result` 来设置。并非所有事件都有结果；带有结果的事件将用 `@HasResult` 进行注释。

重要

不同的事件可能以不同的方式处理结果，在使用事件的结果之前请参阅事件的JavaDoc。

4.3.4 事件处理优先级

事件处理方法（用 `@SubscribeEvent` 标记）具有优先级。你可以通过设置注释的 `priority` 值来安排事件处理方法的优先级。优先级可以是 `EventPriority` 枚举的任何值（ `HIGHEST` 、 `HIGH` 、 `NORMAL` 、 `LOW` 和 `LOWEST` ）。优先级为 `HIGHEST` 的事件处理器首先执行，然后按降序执行，直到最后执行的 `LOWEST` 为止。

4.3.5 子事件

许多事件本身都有不同的变体。这些变体事件可以不尽相同，但都基于一个共同的因素（例如 `PlayerEvent` ），也可以是具有多个阶段的事件（例如 `PotionBrewEvent` ）。请注意，如果你监听父类事件，你的事件处理方法也将收到其所有子类事件。

4.3.6 模组事件总线

模组事件总线主要用于监听模组应该初始化的生命周期事件。模组总线上的每个事件类型都需要实现 `IModBusEvent` 。其中许多事件也是并行运行的（多线程——译者注），因此多个模组可以同时被初始化。这意味着你不能在这些事件中直接执行来自其他模组的代码。为此，请使用 `InterModComms` 系统。

以下是在模组事件总线上的模组初始化期间调用的四个最常用的生命周期事件：

- `FMLCommonSetupEvent`
- `FMLClientSetupEvent` 和 `FMLDedicatedServerSetupEvent`
- `InterModEnqueueEvent`
- `InterModProcessEvent`

注意

`FMLClientSetupEvent` 和 `FMLDedicatedServerSetupEvent` 仅在各自的分发版本（物理端——译者注）上调用。

这四个生命周期事件都是并行运行的，因为它们都是 `ParallelDispatchEvent` 的子类。如果你想在任何 `ParallelDispatchEvent` 期间在主线程上运行运行代码，可以使用 `#enqueueWork` 来执行此操作。

除了生命周期事件之外，还有一些在模组事件总线上触发的杂项事件，你可以在其中注册、设置或初始化各种事情。与生命周期事件相比，这些事件中的大多数不是并行运行的。举几个例子：

- `RegisterColorHandlersEvent`
- `ModelEvent$BakingCompleted`
- `TextureStitchEvent`
- `RegisterEvent`

一个很好的经验法则是：当事件应该在模组初始化期间处理时，就在模组事件总线上触发事件。

4.4 模组生命周期

在模组加载过程中，各种生命周期事件在模组特定的事件总线上触发。在这些事件期间许多操作被执行，例如注册对象、准备数据生成或与其他模组通信。

事件监听器应使用 `@EventBusSubscriber(bus = Bus.MOD)` 或在模组构造函数中被注册：

```
@Mod.EventBusSubscriber(modid = "mymod", bus = Mod.EventBusSubscriber.Bus.MOD)
public class MyModEventSubscriber {
  @SubscribeEvent
  static void onCommonSetup(FMLCommonSetupEvent event) { ... }
}

@Mod("mymod")
public class MyMod {
  public MyMod() {
    FMLModLoadingContext.get().getModEventBus().addListener(this::onCommonSetup);
  }

  private void onCommonSetup(FMLCommonSetupEvent event) { ... }
}
```

> **警告**
>
> 大多数生命周期事件都是并行触发的（多线程——译者注）：所有模组都将同时接收相同的事件。
>
> 模组必须注意线程安全，就像调用其他模组的API或访问原版系统一样。延迟代码，以便稍后通过 `ParallelDispatchEvent#enqueueWork` 执行。

4.4.1 注册表事件

注册表事件是在模组实例构造之后激发的。注册表事件有两种： `NewRegistryEvent` 和 `RegisterEvent` 。这些事件在模组加载期间同步触发。

`NewRegistryEvent` 允许模组开发者使用 `RegistryBuilder` 类注册自己的自定义注册表。

`RegisterEvent` 用于将对象注册到注册表中。每个注册表都会触发该事件。

4.4.2 数据生成

如果游戏被设置为运行数据生成器，那么 `GatherDataEvent` 将是最后一个触发的事件。此事件用于将模组的数据提供者注册到其关联的数据生成器。此事件也是同步触发的。

4.4.3 通用初始化

`FMLCommonSetupEvent` 用于物理客户端和物理服务端通用的操作，例如注册Capability。

4.4.4 单端初始化

单端初始化事件在其各自的物理端触发：物理客户端上触发 `FMLClientSetupEvent` ，dedicated服务端上触发 `FMLDedicatedServerSetupEvent` 。这就是应该进行各物理端特定的初始化的地方，例如注册客户端键盘绑定。

4.4.5 InterModComms

这是模组间可以相互通信以实现跨模组兼容性的地方。有两个相关的事件：`InterModEnqueueEvent` 和 `InterModProcessEvent` 。

`InterModComms` 是负责为模组间交换消息的类。其方法在生命周期事件期间可以安全调用，因为它有 `ConcurrentMap` 支持。

在 `InterModEnqueueEvent` 期间，使用 `InterModComms#sendTo` 以向不同的模组发送消息。这些方法接收所发消息的目的模组的mod id、与消息数据相关的键以及持有消息数据的Supplier。此外，还可以指定消息的发送者，但默认情况下，它将是调用者的mod id。

之后在 `InterModProcessEvent` 期间，使用 `InterModComms#getMessages` 获取所有接收到的消息的Stream。提供的mod id几乎总是先前调用发送消息方法的模组的mod id。此外，可以指定一个Predicate来对消息键进行过滤。这将返回一个带有 `IMCMessages` 的Stream，其中包含数据的发送方、数据的接收方、数据键以及所提供的数据本身。

> **注意**
>
> 还有另外两个生命周期事件：`FMLConstructModEvent`，在模组实例构造之后但在`RegisterEvent`之前直接触发；`FMLLoadCompleteEvent`，在`InterModComms`事件之后触发，用于模组加载过程完成时。

4.5 资源

资源是游戏使用的额外数据，存储在数据文件中，而不是代码中。Minecraft有两个主要的资源系统：一个在逻辑客户端上，用于模型、纹理和本地化等视觉效果，称为 `assets` （资源），另一个在用于游戏的逻辑服务端上，如配方和战利品表，称为 `data` （数据）。资源包（Resource pack）控制前者，而数据包（Datapack）控制后者。

在默认的模组开发工具包中，assets和data目录位于项目的 `src/main/resources` 目录下。

如果启用了多个资源包或数据包，它们会被合并。通常，堆栈顶部包中的文件会覆盖下面的文件；但是，对于某些文件，例如本地化文件和标签，数据实际上是按内容合并的。模组在其 `resources` 目录中定义资源和数据包，但它们被视为“模组资源”包的子集。不能禁用模组资源包，但它们可以被其他资源包覆盖。可以使用原版的 `/datapack` 命令禁用模组数据包。

所有资源都应该有遵循蛇形命名法（Snake Case）的路径和文件名（小写，使用“_”表示单词边界），这在1.11及更高版本中得到了强制执行。

4.5.1 ResourceLocation

Minecraft使用 `ResourceLocation` 识别资源。 `ResourceLocation` 包含两部分：命名空间和路径。它通常指向 `assets/<namespace>/<ctx>/<path>` 处的资源，其中 `ctx` 是特定于上下文的路径片段，取决于 `ResourceLocation` 的使用方式。当从字符串中写入/读取为 `ResourceLocation` 时，它被视为 `<namespace>:<path>` 。如果省略了 `<namespace>:` ，那么当字符串被读取为 `ResourceLocation` 时，命名空间将始终默认为 `minecraft` 。模组应该将其资源放入与其mod id同名的命名空间中（例如，id为 `examplemod` 的模组应该分别将其资源放置在 `assets/examplemod` 和 `data/examplemod` 中，指向这些文件的 `ResourceLocation` 看起来像 `examplemod:<path>` 。）。这不是要求，并且在某些情况下，可能希望使用不同的（或者甚至不止一个）命名空间。 `ResourceLocation` 也在资源系统之外使用，因为它们恰好是唯一标识对象（例如[注册表][]）的好方法。

4.6 国际化与本地化

国际化（Internationalization），简称I18n，是一种设计代码的方式，以便不需要进行任何更改即可适应各种语言。本地化（Localization）是使显示的文本适应用户语言的过程。

I18n是使用 翻译键 来实现的。翻译键是一个字符串，用于指定一段不使用特定语言的可显示文本。例如，`block.minecraft.dirt` 是引用泥土方块名称的翻译键。这样，可显示文本可被引用，而不必考虑特定的语言。这些代码不需要任何更改即可适应新的语言。

本地化将在游戏的语言设置中进行。在Minecraft客户端中，语言环境由语言设置指定。在dedicated服务端上，唯一支持的语言设置是 `en_us` 。可用语言地区的列表可以在Minecraft Wiki上找到。

4.6.1 语言文件

语言文件由 `assets/[namespace]/lang/[locale].json` 定位（例如， `examplemod` 提供的所有美国英语翻译都在 `assets/examplemod/lang/en_us.json` 中）。文件格式只是从翻译键到值的json映射。文件必须使用UTF-8编码。可以使用转换器将旧的.lang文件转换为json。

```
{
  "item.examplemod.example_item": "Example Item Name",
  "block.examplemod.example_block": "Example Block Name",
  "commands.examplemod.examplecommand.error": "Example Command Errored!"
}
```

4.6.2 对方块和物品的用法

Block、Item和其他一些Minecraft类都内置了用于显示其名称的翻译键。这些转换键是通过重写 `#getDescriptionId` 指定的。Item还具有 `#getDescriptionId(ItemStack)` ，重写该方法后可以根据所给ItemStack NBT提供不同的翻译键。

默认情况下， `#getDescriptionId` 将返回以 `block.` 或 `item.` 为前缀的方块或物品的注册表名称，冒号由句点代替。默认情况下， `BlockItem` 覆盖此方法以获取其对应的 `Block` 的翻译密钥。例如，ID为 `examplemod:example_item` 的物品实际上需要语言文件中的以下行：

```
{
  "item.examplemod.example_item": "Example Item Name"
}
```

注意

翻译键的唯一目的是国际化。不要把它们用于代码的逻辑处理部分。请改用注册表名称。

4.6.3 本地化相关方法

警告

一个常见的问题是让服务端为客户端进行本地化。服务端只能在自己的语言设置中进行本地化，这不一定与所连接的客户端的语言设置相匹配。

为了尊重客户端的语言设置，服务端应该让客户端使用 `TranslatableComponent` 或其他保留语言中性翻译键的方法在自己的语言设置中本地化文本。

`net.minecraft.client.resources.language.I18n` (仅客户端)

这个**I18n**类仅在**Minecraft**客户端上有效！它旨在由仅在客户端上运行的代码使用。尝试在服务端上使用它会引发异常并崩溃。

- `get(String, Object...)` 使用格式采取客户端的语言设置进行本地化。第一个参数是翻译键，其余的是 `String.format(String, Object...)` 的格式化参数。

`TranslatableContents`

`TranslatableContents` 是一个经过惰性的本地化和格式化的 `ComponentContents` 。它在向玩家发送消息时非常有用，因为它将在玩家自己的语言设置中自动本地化。

`TranslatableContents(String, Object...)` 构造函数的第一个参数是翻译键，其余参数用于格式化。唯一支持的格式说明符是 `%s` 和 `%1$s` 、 `%2$s` 、 `%3$s` 等。格式化参数可能是将插入到格式化结果文本中并保留其所有属性的 `Component` 。

通过传入 `TranslatableContents` 的参数，可以使用 `Component#translatable` 创建 `MutableComponent` 。它也可以使用 `MutableComponent#create` 通过传入 `ComponentContents` 本身来创建。

`TextComponentHelper`

- `createComponentTranslation(CommandSource, String, Object...)` 根据接收者创建本地化并格式化的 `MutableComponent` 。如果接收者是一个原版客户端，那么本地化和格式化就很容易完成。如果没有，本地化和格式化将使用包含 `TranslatableContents` 的 `Component` 惰性地进行。只有当服务端允许原版客户端连接时，这才有用。

5. 方块

5.1 方块

显然，方块是Minecraft世界的关键。它们构成了所有的地形、结构和机器。如果你有兴趣制作一个模组，那么你必然可能会想添加一些方块。本页将指导你创建方块，以及你可以使用它们做的一些事情。

5.1.1 创建一个方块

基础方块

对于不需要特殊功能的简单方块（比如圆石、木板等），不必自定义一个类。你可以通过使用 `BlockBehaviour$Properties` 对象实例化 `Block` 类来创建一个方块。该 `BlockBehaviour$Properties` 对象可以调用 `BlockBehaviour$Properties#of` 创建，并且可以通过调用其方法进行自定义。例如：

- `strength` - 硬度控制着断块所需的时间。它是一个任意值。作为参考，石头的硬度为1.5，泥土的硬度为0.5。如果该方块不能被破坏，则应使用-1.0的硬度， `Blocks#BEDROCK` 的定义是一个例子。抗性控制块的防爆性。作为参考，石头的抗性为6.0，泥土的抗性为0.5。
- `sound` - 控制方块在点击、破坏或放置时发出的音效。其需要一个 `SoundType` 参数，请参阅音效页面了解更多详细信息。
- `lightLevel` - 控制方块的亮度。其接受一个带有 `BlockState` 参数的函数，该函数返回从0到15的某一个值。
- `friction` - 控制方块的动摩擦系数。作为参考，冰的动摩擦系数为0.98。

所有这些方法都是可链接的，这意味着你可以串联地调用它们。有关此方面的示例，请参见 `Blocks` 类。

> **注意**
>
> `CreativeModeTab` 未针对方块定义setter。如果方块有与之关联的物品（例如 `BlockItem` ），则现在由 `CreativeModeTabEvent$BuildContents` 处理。此外，也没有针对翻译键的setter，因为它现在是从注册表名称生成的。

进阶方块

当然，上面只允许创建非常基本的方块。如果你想添加一些功能，比如玩家交互，那么需要一个自定义的方块类。然而，`Block` 类有很多方法，并且不幸的是，并不是每一个方法都能在这里用文档完全表述。请参阅本节中的其余页面，以了解你可以对方块进行的操作。

5.1.2 注册一个方块

方块必须经过注册后才能发挥作用。

重要

存档中的方块和物品栏中的“方块”是非常不同的东西。存档中的方块由 `BlockState` 表示，其行为由一个 `Block` 类的实例定义。同时，物品栏中的物品是由 `Item` 控制的 `ItemStack` 。作为 `Block` 和 `Item` 二者之间的桥梁，有一个 `BlockItem` 类。 `BlockItem` 是 `Item` 的一个子类，它有一个字段 `block` ，其中包含对它所代表的 `Block` 的引用。 `BlockItem` 将“方块”的一些行为定义为物品，例如右键单击如何放置方块。存在一个没有其 `BlockItem` 的 `Block` 也是可能的。（例如 `minecraft:water` 是一个方块，但不是一个物品。因此，不可能将其作为一个物品保存在物品栏中。）

当一个方块被注册时，也仅仅意味着一个方块被注册了。该方块不会自动具有 `BlockItem` 。要为块创建基本的 `BlockItem` ，应该将 `BlockItem` 的注册表名称设置为其 `Block` 的注册表名称。 `BlockItem` 的自定义子类也可以使用。一旦为方块注册了 `BlockItem` ，就可以使用 `Block#asItem` 来获取它。如果该方块没有 `BlockItem` ， `Block#asItem` 将返回 `Items#AIR` ，因此，如果你不确定你正在使用的方块是否有 `BlockItem` ，请检查其 `Block#asItem` 是否返回 `Items#AIR` 。

选择性地注册方块

在过去，有一些模组允许用户在配置文件中禁用方块/物品。但是，你不应该这样做。允许注册的方块数量没有限制，所以请在你的模组中注册所有方块！如果你想通过配置文件禁用一个方块，你应该禁用其配方。如果要禁用创造模式物品栏中的方块，请在 `CreativeModeTabEvent$BuildContents` 中构建内容时使用 `FeatureFlag` 。

5.1.3 延伸阅读

有关方块属性的信息，例如用于栅栏、墙等原版方块的属性，请参见方块状态部分。

5.2 方块状态

5.2.1 旧版本的行为

在Minecraft 1.7及以前的版本中，需要存储没有BlockEntity的位置或状态数据的方块使用元数据（**metadata**）。元数据是与方块一起存储的额外数字，允许方块不同的旋转、朝向，甚至完全独立的行为。

然而，元数据系统是令人费解且有限度的，因为它只存储为方块ID旁边的一个数字，离开了代码中注释的内容后便没有任何意义。例如，要实现可以面向某个方向并且位于方块空间（例如楼梯）的上半部分或下半部分的方块，下列操作被执行：

```
switch (meta) {
  case 0: { ... } // 面向南方且位于方块的下半部分
  case 1: { ... } // 面向南方且位于方块的上半部分
  case 2: { ... } // 面向北方且位于方块的下半部分
  case 3: { ... } // 面向北方且位于方块的上半部分
  // ... etc. ...
}
```

因为这些数字本身没有任何意义，所以除非能够访问源代码和注释，否则没有人能够知道它们代表什么。

5.2.2 状态简介

在Minecraft 1.8及以上版本中，元数据系统和方块ID系统被弃用，最终被方块状态系统取代。方块状态系统从方块的其他行为中抽象出方块属性的细节。

方块的每个属性都由 `Property<?>` 的一个实例来描述。方块属性的示例包括乐器（`EnumProperty<NoteBlockInstrument>`）、朝向（`DirectionProperty`）、充能状态（`Property<Boolean>`）等。每个属性都具有由 `Property<T>` 参数化后的类型 `T` 的值。

可以构建从 `Block` 和 `Property<?>` 的Map到与它们相关联的值的唯一的对。这个唯一的对被称为 `BlockState`。

以前无意义的元数据值系统被更容易解释和处理的方块属性系统所取代。以前，朝向东方并充能或按下的石头按钮由“带有元数据 `9` 的 `minecraft:stone_button`”表示。现在，其用“`minecraft:stone_button[facing=east,powered=true]`”来表示。

5.2.3 方块状态系统的正确用法

`BlockState` 系统是一个灵活而强大的系统，但它也有局限性。`BlockState` 是不可变的，其属性的所有组合都是在游戏启动时生成的。这意味着拥有一个具有许多属性和可能值的 `BlockState` 会减慢游戏的加载速度，并让任何试图理解你的方块逻辑的人感到困惑。

并非所有方块和情况都需要使用 `BlockState`；只有方块的最基本属性才应该被放入 `BlockState`，而任何其他情况都最好有一个 `BlockEntity` 或是一个区分开的 `Block`。要始终考虑是否确实需要出于你的目的而使用方块状态。

注意

一个很好的经验法则是：如果它有不同的名称，那么它应该是一个单独的方块。

一个案例是制作椅子方块：椅子的朝向应该是一个属性，而不同类型的木材应该被分成不同的块。朝向东方的“橡木椅子”（`oak_chair[facing=east]`）与朝向西方的“云杉椅子”（`oak_chair[facing=east]`）不同。

5.2.4 实现方块状态

在你的Block类中，为你的方块所拥有的所有属性创建或引用 `static final Property<?>` 对象。你尽可以创建自己的 `Property<?>` 实现，但本文没有介绍实现的方法。原版代码提供了几个方便的实现：

- `IntegerProperty`
  - 实现了 `Property<Integer>` 。定义具有整数值的一个属性。
  - 通过调用 `IntegerProperty#create(String propertyName, int minimum, int maximum)` 创建。
- `BooleanProperty`
  - 实现了 `Property<Boolean>` 。定义具有 `true` 或 `false` 值的一个属性。
  - 通过调用 `BooleanProperty#create(String propertyName)` 创建。
- `EnumProperty<E extends Enum<E>>`
  - 实现了 `Property<E>` 定义具有某一枚举类的值的一个属性。
  - 通过调用 `EnumProperty#create(String propertyName, Class<E> enumClass)` 创建。
  - 也能够仅使用枚举值的一个子集（例如16个 `DyeColor` 中的4个）。请参阅 `EnumProperty#create` 的重载。
- `DirectionProperty`
  - 这是 `EnumProperty<Direction>` 的一个便利实现。
  - Several convenience predicates are also provided. For example, to get a property that represents the cardinal directions, call `DirectionProperty.create("<name>", Direction.Plane.HORIZONTAL)`; to get the X directions, `DirectionProperty.create("<name>", Direction.Axis.X)`.
  - 也提供了一些便利的Predicate。例如，要获得一个表示基本方向的属性，请调用 `DirectionProperty.create("<name>", Direction.Plane.HORIZONTAL)`；要获得仅X方向的，请调用 `DirectionProperty.create("<name>", Direction.Axis.X)` 。

类 `BlockStateProperties` 包含共有的原版属性，应该尽可能先使用或引用这些属性，而不是创建自己的属性。

当你拥有所需的 `Property<>` 对象时，请重写你的Block类中的 `Block#createBlockStateDefinition(StateDefinition$Builder)` 。在该方法中，使用你希望这个方块具有的每个 `Property<?>` 作为参数调用 `StateDefinition$Builder#add(...);` 。

每个方块也将有一个自动为你选择的“默认”状态。你可以通过从构造函数中调用 `Block#registerDefaultState(BlockState)` 方法来更改此“默认”状态。放置方块后，它将变为此“默认”状态。如 `DoorBlock` 的一个案例：

```
this.registerDefaultState(
  this.stateDefinition.any()
    .setValue(FACING, Direction.NORTH)
    .setValue(OPEN, false)
    .setValue(HINGE, DoorHingeSide.LEFT)
    .setValue(POWERED, false)
    .setValue(HALF, DoubleBlockHalf.LOWER)
);
```

如果你希望更改放置方块时使用的 `BlockState` ，可以重写 `Block#getStateForPlacement(BlockPlaceContext)` 。例如，这可以用来根据玩家放置方块时所站的位置来设置方块的方向。

由于 `BlockState` 是不可变的，并且其属性的所有组合都是在游戏启动时生成的，因此调用 `BlockState#setValue(Property<T>, T)` 将只需转到 `Block` 的 `StateHolder` 并请求具有你所需的一系列值的 `BlockState` 。

因为所有可能的 `BlockState` 都是在启动时生成的，所以你可以自由地使用引用相等运算符（ `==` ）来检查两个 `BlockState` 是否相等。

5.2.5 使用 BlockState

你可以通过调用 `BlockState#getValue(Property<?>)` 来获取某个属性的值，并将要获取值的属性传递给它。如果你想获得一个具有不同的一系列值的 `BlockState` ，只需使用该属性及其值调用 `BlockState#setValue(Property<T>, T)` 。

你可以使用 `Level#setBlockAndUpdate(BlockPos, BlockState)` 和 `Level#getBlockState(BlockPos)` 在存档中获取并放置 `BlockState` 。如果你正在放置一个 `Block` ，请调用 `Block#defaultBlockState()` 以获得“默认”状态，并使用对 `BlockState#setValue(Property<T>, T)` 的后续调用，如上所述，以保存所需状态。

6. 物品

6.1 物品

与方块一样，物品也是大多数模组的关键组成部分。方块在构成了你身边的存档的同时，物品也存在于物品栏中。

6.1.1 创建一个物品

基础物品

对于不需要特殊功能的简单物品（比如木棍或糖），不必自定义一个类。你可以通过使用 `Item$Properties` 对象实例化 `Item` 类来创建一个物品。这个 `Item$Properties` 对象可以通过调用其构造函数生成并通过调用其方法进行自定义。例如：

方法	描述
`requiredFeatures`	设置在所添加到的 `CreativeModeTab` 中查看此物品所需的 `FeatureFlag` 。
`durability`	设置该物品的最大耐久。如果超过 `0` ，两个物品属性“damaged”和“damage”会被添加。
`stacksTo`	设置最大物品栈大小。你不能拥有一件既有耐久又可堆叠的物品。
`setNoRepair`	使此物品无法修复，即使它是有耐久的。
`craftRemainder`	设置该物品的容器物品，即熔岩桶在使用后将空桶还给你的方式。

上面的方法是可链接的，这意味着它们 `return this` 以便于串行调用它们。

进阶物品

如上所述设置物品属性的方式仅适用于简单物品。如果你想要更复杂的物品，你应该继承 `Item` 类并重写其方法。

6.1.2 CreativeModeTabEvent

可以通过模组事件总线上的 `CreativeModeTabEvent$BuildContents` 将物品添加到 `CreativeModeTab` 。可以通过 `#accept` 添加物品，而无需任何其他配置。

```
// 已在模组事件总线上注册
// 假设我们有一个名为ITEM的RegistryObject<Item>和一个名为BLOCK的RegistryObject<Block>
@SubscribeEvent
public void buildContents(CreativeModeTabEvent.BuildContents event) {
  // 添加到ingredients创造模式物品栏
  if (event.getTab() == CreativeModeTabs.INGREDIENTS) {
    event.accept(ITEM);
    event.accept(BLOCK); // 接受一个ItemLike，假设方块已注册其物品
  }
}
```

你还可以通过 `FeatureFlagSet` 中的 `FeatureFlag` 或一个用于确定玩家是否有权查看管理员创造模式物品栏的boolean值来启用或禁用物品。

自定义创造模式物品栏

可以通过模组事件总线上的 `CreativeModeTabEvent$Register#registerCreativeModeTab` 创建自定义的 `CreativeModeTab` 。这需要使用物品栏页的名称和一个构建器的Consumer。此外，还可以提供 `ResourceLocation` 或 `CreativeModeTab` 的列表，以确定物品栏页的位置。

```
// 已在模组事件总线上注册
// 假设我们有一个名为ITEM的RegistryObject<Item>和一个名为BLOCK的RegistryObject<Block>
@SubscribeEvent
public void buildContents(CreativeModeTabEvent.Register event) {
  event.registerCreativeModeTab(new ResourceLocation(MOD_ID, "example"), builder ->
    // 设置所要展示的页的名称
    builder.title(Component.translatable("item_group." + MOD_ID + ".example"))
    // 设置页图标
    .icon(() -> new ItemStack(ITEM.get()))
    // 为物品栏页添加默认物品
    .displayItems((params, output) -> {
      output.accept(ITEM.get());
      output.accept(BLOCK.get());
    })
  );
}
```

6.1.3 注册一个物品

物品必须经过注册后才能发挥作用。

6.2 BlockEntityWithoutLevelRenderer

`BlockEntityWithoutLevelRenderer` 是一种处理物品的动态渲染的方法。这个系统比旧的 `ItemStack` 系统简单得多，旧的 `ItemStack` 系统需要 `BlockEntity` ，并且不允许访问 `ItemStack` 。

6.2.1 使用BlockEntityWithoutLevelRenderer

BlockEntityWithoutLevelRenderer允许你使用

`public void renderByItem(ItemStack itemStack, ItemDisplayContext ctx, PoseStack poseStack, MultiBufferSource bufferSource, int combinedLight, int co`

来渲染物品。

为了使用BEWLR， `Item` 必须首先满足其模型的 `BakedModel#isCustomRenderer` 返回true。如果没有，它将使用默认的 `ItemRenderer#getBlockEntityRenderer` 。一旦返回true，将访问该Item的BEWLR进行渲染。

注意

如果 `Block#getRenderShape` 设置为 `RenderShape#ENTITYBLOCK_ANIMATED` ， `Block` 也会使用BEWLR进行渲染。

若要设置物品的BEWLR，必须在 `Item#initializeClient` 中使用 `IClientItemExtensions` 的一个匿名实例。在该匿名实例中，应重写 `IClientItemExtensions#getCustomRenderer` 以返回你的BEWLR的实例：

```
// 在你的物品类中
@Override
public void initializeClient(Consumer<IClientItemExtensions> consumer) {
  consumer.accept(new IClientItemExtensions() {

    @Override
    public BlockEntityWithoutLevelRenderer getCustomRenderer() {
      return myBEWLRInstance;
    }
  });
}
```

重要

每个模组都应该只有一个自定义BEWLR的实例。

这就行了，使用BEWLR不需要额外的设置。

7. 网络

7.1 网络

服务端与客户端之间的通信是成功实现模组的中流砥柱。

网络通信有两个主要目标:

1. 确保客户端视图与服务端视图“同步”
    - 坐标（X，Y，Z）处的花刚刚生长
2. 为客户端提供一种方法，告诉服务端玩家发生了变化
    - 玩家按下了一个按键

实现这些目标的最常见方法是在客户端和服务端之间传递消息。这些消息通常是结构化的，包含特定排列的数据，以便于发送和接收。

Forge提供了多种技术来促进通信，这些技术大多建立在netty之上。

对于一个新模组来说，最简单的当是SimpleImpl，在这里，网络系统的大部分复杂性都被抽象掉了。它使用消息和处理器样式的系统。

7.2 SimpleImpl

SimpleImpl是围绕 `SimpleChannel` 类的数据包系统的名称。使用此系统是迄今为止在客户端和服务端之间发送自定义数据的最简单方法。

7.2.1 快速入门

首先，你需要创建 `SimpleChannel` 对象。我们建议你在单独的类中执行此操作，可能类似于 `ModidPacketHandler` 。将 `SimpleChannel` 创建为此类中的静态字段，如下所示：

```
private static final String PROTOCOL_VERSION = "1";
public static final SimpleChannel INSTANCE = NetworkRegistry.newSimpleChannel(
  new ResourceLocation("mymodid", "main"),
  () -> PROTOCOL_VERSION,
  PROTOCOL_VERSION::equals,
  PROTOCOL_VERSION::equals
);
```

第一个参数是通道的名称。第二个参数是返回当前网络协议版本的 `Supplier<String>` 。第三个和第四个参数分别是 `Predicate<String>` ，分别检查传入的连接协议版本是否与客户端或服务端网络兼容。在这里，我们只需直接与 `PROTOCOL_VERSION` 字段进行比较，这意味着客户端和服务端 `PROTOCOL_VERSION` 必须始终匹配，否则FML将拒绝登录。

7.2.2 版本检查器

如果你的模组不要求另一端拥有特定的网络通道，或者根本不要求对方是Forge实例，你应该注意正确定义你的版本兼容性检查器（ `Predicate<String>` 参数），以处理版本检查器可以接收的其他“元版本”（在 `NetworkRegistry` 中定义）。这些是：

- `ABSENT` - 如果该通道在另一个端点上丢失。请注意，在这种情况下，端点仍然是Forge端点，并且可能具有其他模组。
- `ACCEPTVANILLA` - 如果端点是原版（或非Forge）端点（如Fabric——译者注）。

对两者返回 `false` 意味着该通道必须存在于另一端上。如果你只是复制上面的代码，这就是它的作用。请注意，在列表ping兼容性检查期间也会使用这些值，该检查负责在多人服务器选择屏幕中显示绿色复选框/红叉。

7.2.3 注册数据包

接下来，我们必须声明要发送和接收的消息类型。这是使用 `INSTANCE#registerMessage` 完成的，它接受5个参数：

- 第一个参数是数据包的鉴别器。这是数据包的每个通道的唯一ID。我们建议你使用本地变量来保存ID，然后使用 `id++` 调用registerMessage。这将保证100%的唯一ID。
- 第二个参数是实际的数据包类 `MSG` 。
- 第三个参数是 `BiConsumer<MSG, FriendlyByteBuf>` ，负责将消息编码到所提供的 `FriendlyByteBuf` 中。
- 第四个参数是 `Function<FriendlyByteBuf, MSG>` ，负责从所提供的 `FriendlyByteBuf` 中解码消息。
- 最后一个参数是负责处理消息本身的 `BiConsumer<MSG, Supplier<NetworkEvent.Context>>` 。

最后三个参数可以是Java中静态方法或实例方法的方法引用。请记住，实例方法 `MSG#encode(FriendlyByteBuf)` 仍然满足 `BiConsumer<MSG, FriendlyByteBuf>` ； `MSG` 只不过成为隐含的第一个自变量。

7.2.4 处理数据包

在数据包处理器中，有几件事需要强调。数据包处理器同时具有对其可用消息对象和网络上下文。该上下文允许访问发送数据包的玩家（如果在服务端上），并允许一种方式将线程安全工作排入队列。

```
public static void handle(MyMessage msg, Supplier<NetworkEvent.Context> ctx) {
  ctx.get().enqueueWork(() -> {
    // 要求线程安全的工作（大多数工作）
    ServerPlayer sender = ctx.get().getSender(); // 发送该数据包的客户端
    // 处理一些事情
  });
  ctx.get().setPacketHandled(true);
}
```

从服务端发送到客户端的数据包应在另一个类中进行处理，并通过 `DistExecutor#unsafeRunWhenOn` 进行包装。

```
// 在Packet类中
public static void handle(MyClientMessage msg, Supplier<NetworkEvent.Context> ctx) {
  ctx.get().enqueueWork(() ->
    // 确保其仅在物理客户端上执行
    DistExecutor.unsafeRunWhenOn(Dist.CLIENT, () -> () -> ClientPacketHandlerClass.handlePacket(msg, ctx))
  );
  ctx.get().setPacketHandled(true);
}

// 在ClientPacketHandlerClass中
public static void handlePacket(MyClientMessage msg, Supplier<NetworkEvent.Context> ctx) {
  // 处理一些事情
}
```

请注意 `#setPacketHandled` 的存在，它用于告诉网络系统该数据包已成功完成处理。

警告

从Minecraft 1.8开始，默认情况下在网络线程上处理数据包。

这意味着你的处理器 不 能直接与大多数游戏对象交互。Forge提供了一种方便的方法，可以通过提供的 `NetworkEvent$Context` 在主线程上执行代码。只需调用 `NetworkEvent$Context#enqueueWork(Runnable)`，它将在下一次有机会时调用主线程上的给定 `Runnable`。

警告

在服务端上处理数据包时要采取防御措施。客户端可能试图通过发送意外数据来对数据包处理过程施压。

一个常见的问题是易受任意区块生成的攻击。当服务端信任客户端发送的方块位置来访问方块和块方实体时，通常会发生这种情况。当访问存档中的未加载区域中的方块和方块实体时，服务端将会要么生成要么从磁盘加载该区域，然后立即将其写入磁盘。利用这一点，可以在不留下痕迹的情况下对服务端的性能和存储空间造成灾难性破坏。

为了避免这个问题，一个普遍的经验法则是，仅访问 `Level#hasChunkAt` 为true的方块和方块实体。

7.2.5 发送数据包

向服务端发送

只有一种方法可以将数据包发送到服务端。这是因为客户端一次只能连接到一个服务端。要做到这一点，我们必须再次使用前面定义的 `SimpleChannel` 。只需调用 `INSTANCE.sendToServer(new MyMessage())` 。消息将被发送到对应其类型的处理器（如果存在）。

向客户端发送

数据包可以使用 `SimpleChannel` 直接发送到客户端：
`HANDLER.sendTo(new MyClientMessage(), serverPlayer.connection.getConnection(), NetworkDirection.PLAY_TO_CLIENT)` 。但是，这可能很不方便。Forge有一些可以使用的便利功能：

```
// 向一位玩家发送
INSTANCE.send(PacketDistributor.PLAYER.with(serverPlayer), new MyMessage());

// 向正在追踪该存档某个区块的所有玩家发送
INSTANCE.send(PacketDistributor.TRACKING_CHUNK.with(levelChunk), new MyMessage());

// 向所有已连接的玩家发送
INSTANCE.send(PacketDistributor.ALL.noArg(), new MyMessage());
```

还有其他类型的 `PacketDistributor` 可用；有关更多详细信息，请查看 `PacketDistributor` 类的文档。

7.3 实体

除了常规的网络消息之外，Forge还提供了各种其他系统来处理同步实体数据。

7.3.1 生成数据

一般来说，由模组编写的实体的生成是由Forge单独处理的。

> **注意**
>
> 这意味着简单地继承一个原版实体类可能不会继承它的所有行为。你可能需要自己实施某些原版行为。

你可以通过实现以下接口向Forge发送的生成数据包添加额外的数据。

IEntityAdditionalSpawnData

如果你的实体具有客户端所需的数据，但不会随时间变化，则可以使用此接口将其添加到实体生成数据包中。`#writeSpawnData` 和 `#readSpawnData` 控制如何将数据编码到网络缓冲区/从网络缓冲区解码数据。

7.3.2 动态数据

数据参数

这是用于将实体数据从服务端同步到客户端的主要原版系统。因此，可以参考一些原版的例子。

首先，对于要保持同步的数据，你需要一个 `EntityDataAccessor<T>` 。这应该存储为你的实体类中的 `static final` 字段，通过调用 `SynchedEntityData#defineId` 并传递实体类和该类型数据的序列化器来获得。可用的序列化器实现可以在 `EntityDataSerializers` 类中的静态常量找到。

> **警告**
>
> 你应该 只 在相应实体的类 中为自己的实体创建数据参数。 向并非你所控制的实体添加参数可能会导致用于通过网络发送数据的ID不同步，从而导致难以调试的崩溃。

然后，重写 `Entity#defineSynchedData` 并为每个数据参数调用 `this.entityData.define(...)` ，传递参数和要使用的初始值。请记住始终首先调用 `super` 方法！

然后，你可以通过实体的 `entityData` 实例获取并设置这些值。所做的更改将自动同步到客户端。

8. 方块实体

8.1 方块实体

`BlockEntities` 类似于绑定到某一方块的简化的 `Entities` 。 它们能用于存储动态数据、执行基于游戏刻的任务和动态渲染。原版Minecraft中的一些例子是处理箱子的物品栏、熔炉的熔炼逻辑或信标的区域效果。 模组中存在更高级的示例，例如采石场（如BC）、分拣机（如IC2）、管道（如BC）和显示器（如OC）。（括号内容为译者注。）

注意

`BlockEntities` 并不是万能的解决方案，如果使用错误，它们可能会导致游戏卡顿。 如果可能的话，尽量避免使用。

8.1.1 注册

方块实体是动态创建和删除的，因此它们本身不是注册表对象。

为了创建 `BlockEntity` ，你需要继承 `BlockEntity` 类。这样，另一个对象被替代性地注册以方便创建和引用动态对象的类型。对于 `BlockEntity` ，这些对象被称为 `BlockEntityType` 。

`BlockEntityType` 可以像任何其他注册表对象一样进行注册。若要构造 `BlockEntityType` ，可以通过 `BlockEntityType$Builder#of` 使用其Builder形式。这需要两个参数： `BlockEntityType$BlockEntitySupplier` ，它接受 `BlockPos` 和 `BlockState` 来创建关联 `BlockEntity` 的新实例，以及该 `BlockEntity` 可以附加到的 `Block` 的可变参数。构建该 `BlockEntityType` 是通过调用 `BlockEntityType$Builder#build` 来完成的。其接受一个 `Type` ，表示用于引用某个 `DataFixer` 中的此注册表对象的类型安全引用。由于 `DataFixer` 是用于模组的可选系统，因此其也可用 `null` 代替。

```
// 对于某个类型为DeferredRegister<BlockEntityType<?>>的REGISTER
public static final RegistryObject<BlockEntityType<MyBE>> MY_BE = REGISTER.register("mybe", () -> BlockEntityType.Builder.of(MyBE::r

// 在MyBE（一个BlockEntity的子类）中
public MyBE(BlockPos pos, BlockState state) {
  super(MY_BE.get(), pos, state);
}
```

8.1.2 创建一个 BlockEntity

要创建 `BlockEntity` 并将其附加到 `Block` ， `EntityBlock` 接口必须在你的 `Block` 子类上实现。方法 `EntityBlock#newBlockEntity(BlockPos, BlockState)` 必须实现并返回一个你的 `BlockEntity` 的新实例。

8.1.3 将数据存储到你的 BlockEntity

为了保存数据，请重写以下两个方法：

```
BlockEntity#saveAdditional(CompoundTag tag)

BlockEntity#load(CompoundTag tag)
```

每当包含 `BlockEntity` 的 `LevelChunk` 从标签加载/保存到标签时，都会调用这些方法。 使用它们以读取和写入你的方块实体类的字段。

注意

每当你的数据发生改变时，你需要调用 `BlockEntity#setChanged`；否则，保存存档时可能会跳过包含你的 `BlockEntity` 的 `LevelChunk`。

重要

调用 `super` 方法非常重要！

标签名称 `id`、`x`、`y`、`z`、`ForgeData` 和 `ForgeCaps` 均由 `super` 方法保留。

8.1.4 计时的 BlockEntity

如果你需要一个计时的 `BlockEntity`，例如为了跟踪冶炼过程中的进度，则必须在 `EntityBlock` 中实现并重写另一个方法：`EntityBlock#getTicker(Level, BlockState, BlockEntityType)`。这可以根据用户所处的逻辑端实现不同的计时器，或者只实现一个通用计时器。无论哪种情况，都必须返回 `BlockEntityTicker`。由于这是一个功能性的接口，因此它可以转而采用一个表示计时器的方法：

```
// 在某个Block子类内
@Nullable
@Override
public <T extends BlockEntity> BlockEntityTicker<T> getTicker(Level level, BlockState state, BlockEntityType<T> type) {
  return type == MyBlockEntityTypes.MYBE.get() ? MyBlockEntity::tick : null;
}

// 在MyBlockEntity内
public static void tick(Level level, BlockPos pos, BlockState state, MyBlockEntity blockEntity) {
  // 处理一些事情
}
```

注意

这个方法在每个游戏刻都会调用；因此，你应该避免在这里进行复杂的计算。如果可能的话，你应该每X个游戏刻进行更复杂的计算。（一秒钟内的游戏刻数量可能低于20（二十），但不会更高）

8.1.5 向客户端同步数据

有三种方法可以将数据同步到客户端：在区块加载时同步、在方块更新时同步以及使用自定义网络消息同步。

在**LevelChunk**加载时同步

为此你需要重写

```
BlockEntity#getUpdateTag()

IForgeBlockEntity#handleUpdateTag(CompoundTag tag)
```

同样，这非常简单，第一个方法收集应该发送到客户端的数据，而第二个方法处理这些数据。如果你的 `BlockEntity` 不包含太多数据，你可以使用将数据存储到你的 `BlockEntity` 小节之外的方法。

重要

为方块实体同步过多/无用的数据可能会导致网络拥塞。你应该通过在客户端需要时仅发送客户端需要的信息来优化网络使用。例如，在更新标签中发送方块实体的物品栏通常是没有必要的，因为这可以通过其`AbstractContainerMenu`进行同步。

在方块更新时同步

这个方法有点复杂，但同样，你只需要重写两个或三个方法。 下面是它的一个简易的实现示例：

```
@Override
public CompoundTag getUpdateTag() {
  CompoundTag tag = new CompoundTag();
  //将你的数据写入标签
  return tag;
}

@Override
public Packet<ClientGamePacketListener> getUpdatePacket() {
  // 将从#getUpdateTag得到标签
  return ClientboundBlockEntityDataPacket.create(this);
}

// 可以重写IForgeBlockEntity#onDataPacket。默认地，其将遵从#load。
```

静态构造器 `ClientboundBlockEntityDataPacket#create` 接受：

- 该 `BlockEntity` 。
- 从该 `BlockEntity` 中获取 `CompoundTag` 的可选函数。默认情况下，其使用 `BlockEntity#getUpdateTag` 。

现在，要发送数据包，必须在服务端上发出更新通知。

```
Level#sendBlockUpdated(BlockPos pos, BlockState oldState, BlockState newState, int flags)
```

`pos` 应为你的 `BlockEntity` 的位置。 对于 `oldState` 和 `newState` ，你可以传递那个位置的 `BlockState` 。 `flags` 是一个应含有 `2` 的位掩码（bitmask），其将向客户端同步数据。有关更多信息以及flags的其余信息，参见 `Block` 。flag `2` 与 `Block#UPDATE_CLIENTS` 相同。

使用自定义网络消息同步

这种同步方式可能是最复杂的，但通常是最优化的，因为你可以确保只有需要同步的数据才是真正同步的。在尝试之前，你应该先查看 `Networking` 部分，尤其是 `SimpleImpl` 。一旦你创建了自定义网络消息，你就可以使用 `SimpleChannel#send(PacketDistributor$PacketTarget, MSG)` 将其发送给所有加载了该 `BlockEntity` 的用户。

警告

进行安全检查很重要，当消息到达玩家时，`BlockEntity`可能已经被销毁/替换！你还应该检查区块是否已加载（`Level#hasChunkAt(BlockPos)`）。

8.2 BlockEntityRenderer

`BlockEntityRenderer`（简称 `BER`）用于以静态烘焙模型（JSON、OBJ、B3D等）无法表示的方式渲染方块。方块实体渲染器要求方块具有 `BlockEntity`。

8.2.1 创建一个BER

要创建BER，请创建一个继承自 `BlockEntityRenderer` 的类。它采用一个泛型参数来指定方块的 `BlockEntity` 类。该泛型参数用于BER的 `render` 方法。

对于任意一个给定的 `BlockEntityType`，仅存在一个BER。因此，特定于存档中单个实例的值应该存储在传递给渲染器的方块实体中，而不是存储在BER本身中。例如，如果将逐帧递增的整数存储在BER中，则对于该存档中该类型的每个方块实体也会逐帧递增。

`render`

为了渲染方块实体，每帧都调用此方法。

参数

- `blockEntity`：这是正在渲染的方块实体的实例。
- `partialTicks`：在帧的摩擦过程中，从上一次完整刻度开始经过的时间量。
- `poseStack`：一个栈，包含偏移到方块实体当前位置的四维矩阵条目。
- `bufferSource`：能够访问顶点Consumer的渲染缓冲区。
- `combinedLight`：方块实体上当前亮度值的整数。
- `combinedOverlay`：设置为方块实体的当前overlay的整数，通常为 `OverlayTexture#NO_OVERLAY` 或655,360。

8.2.2 注册一个BER

要注册BER，你必须订阅模组事件总线上的 `EntityRenderersEvent$RegisterRenderers` 事件，并调用 `#registerBlockEntityRenderer`。

9. 游戏特效

9.1 粒子效果

粒子是游戏中的一种效果，用于打磨游戏，以更好地提高沉浸感。由于它们的创建和引用方法，其有用性也需要非常谨慎地对待。

9.1.1 创建一个粒子

粒子被分解为仅用于显示粒子的仅客户端实现和用于引用来自服务端的粒子或同步数据的通用实现。

类	物理端	描述
ParticleType	BOTH	粒子类型定义的注册表对象，用于引用任一端位的粒子
ParticleOptions	BOTH	用于将来自网络或命令的信息同步到相关客户端的数据保持器
ParticleProvider	CLIENT	由 `ParticleType` 注册的工厂，用于从关联的 `ParticleOptions` 构造 `Particle` 。
Particle	CLIENT	要在关联客户端上显示的可渲染逻辑

ParticleType

`ParticleType` 是定义特定粒子类型的注册表对象，并提供对两端位特定粒子的可用引用。因此，每个 `ParticleType` 都必须注册。

每个 `ParticleType` 都有两个参数：一个 `overrideLimiter` ，用于确定粒子是否在不考虑距离的情况下渲染，以及一个 `ParticleOptions$Deserializer` ，用于读取客户端上发送的 `ParticleOptions` 。由于基类 `ParticleType` 是抽象类，因此需要实现一个方法： `#codec` 。其表示如何对与该类型相关的 `ParticleOptions` 进行编码和解码。

> **注意**
>
> `ParticleType#codec` 仅在用于原版实现的生物群系编解码器中使用。

在大多数情况下，不需要将任何粒子数据发送到客户端。对于这些例子，更容易创建 `SimpleParticleType` 的新实例：一个对 `ParticleType` 和 `ParticleOptions` 的实现，除了类型之外，它不向客户端发送任何自定义数据。除了红石粉之外，对于着色和依赖方块/物品的粒子而言，大多数原版实现还使用 `SimpleParticleType` 。

> **重要**
>
> 如果仅在客户端上引用，则生成粒子时 `ParticleType` 非必要。但是，有必要使用 `ParticleEngine` 中的任何预构建逻辑，或者从服务端生成粒子。

ParticleOptions

`ParticleOptions` 表示每个粒子所接收的数据。它还用于发送通过服务端生成的粒子的数据。所有粒子生成方法都接受一个 `ParticleOptions` ，这样它就知道粒子的类型以及与生成方法关联的数据。

`ParticleOptions` 被拆分为三种方法：

方法	描述
getType	获取粒子的类型定义，或 `ParticleType`
writeToNetwork	将粒子数据写入服务端上的缓冲区以发送到客户端
writeToString	将粒子数据写入字符串

这些对象要么是根据需要动态构建的，要么是作为 `SimpleParticleType` 的结果而产生的单体。

ParticleOptions$Deserializer

要在客户端上接收 `ParticleOptions`，或引用命令中的数据，必须通过 `ParticleOptions$Deserializer` 对粒子数据进行反序列化。`ParticleOptions$Deserializer` 中的每个方法都对等 `ParticleOptions` 的编码方法：

方法	ParticleOptions编码器	描述
fromCommand	writeToString	从字符串（通常是从命令）中解码粒子数据。
fromNetwork	writeToNetwork	解码客户端缓冲区中的粒子数据。

当需要发送自定义粒子数据时，此对象会传递到 `ParticleType` 的构造函数中。

Particle

`Particle` 提供将所述数据绘制到屏幕上所需的渲染逻辑。要创建任何 `Particle`，必须实现两个方法：

方法	描述
render	将粒子渲染到屏幕上。
getRenderType	获取粒子的渲染类型。

用于渲染纹理的 `Particle` 的一个常见子类是 `TextureSheetParticle`。虽然需要实现 `#getRenderType`，但无论设置了什么纹理 sprite，都将在粒子的位置进行渲染。

ParticleRenderType

`ParticleRenderType` 是 `RenderType` 的一个变体，它为该类型的每个粒子构造启动和拆卸阶段，然后通过 `Tesselator` 同时渲染所有粒子。粒子可以使用六种不同的渲染类型。

渲染类型	描述
TERRAIN_SHEET	渲染纹理位于可用方块内的粒子。
PARTICLE_SHEET_OPAQUE	渲染纹理不透明且位于可用粒子内的粒子。
PARTICLE_SHEET_TRANSLUCENT	渲染纹理为半透明且位于可用粒子内的粒子。
PARTICLE_SHEET_LIT	与 `PARTICLE_SHEET_OPAQUE` 相同，但不使用粒子着色器。
CUSTOM	提供混合和深度遮罩的设置，但不提供将在 `Particle#render` 中实现的渲染功能。
NO_RENDER	粒子将永远不会渲染。

实现自定义渲染类型将留给读者练习。

ParticleProvider

最后，粒子通常是通过 `ParticleProvider` 创建的。工厂有一个单一的方法 `ParticleProvider`，用于在给定粒子数据、客户端存档、位置和移动增量的情况下创建粒子。由于 `Particle` 不受任何特定 `ParticleType` 的约束，因此可以根据需要在不同的工厂中重复使用。

必须通过订阅模组事件总线上的 `RegisterParticleProvidersEvent` 以注册 `ParticleProvider`。在事件中，可以通过向方法提供工厂实例，通过 `#registerSpecial` 注册工厂。

> **重要**
>
> `RegisterParticleProvidersEvent` 应仅在客户端上调用，因此在某些客户端类中被单端化独立，并被 `DistExecutor` 或 `@EventBusSubscriber` 引用。

ParticleDescription、SpriteSet、以及SpriteParticleRegistration

有三种粒子渲染类型不能使用上述注册方法：`PARTICLE_SHEET_OPAQUE`、`PARTICLE_SHEET_TRANSLUCENT` 和 `PARTICLE_SHEET_LIT`。这是因为这三种粒子渲染类型都使用由 `ParticleEngine` 直接加载的sprite集。因此，所提供的纹理必须通过不同的方法获得和注册。这将假设你的粒子是 `TextureSheetParticle` 的子类型，因为这是该逻辑的唯一原版实现。

要将纹理添加到粒子，必须将一个新的JSON文件添加到 `assets/<modid>/particles`。这被称为 `ParticleDescription`。该文件的名称将代表工厂所附加的 `ParticleType` 的注册表名称。每个粒子JSON都是一个对象。该对象存储单个关键的 `textures`，该键包含 `ResourceLocation` 的一个数组。此处表示的任何 `<modid>:<path>` 纹理都将指向 `assets/<modid>/textures/particle/<path>.png` 处的纹理。

```
{
  "textures": [
    // Will point to a texture located in
    // assets/mymod/textures/particle/particle_texture.png
    "mymod:particle_texture",
    // Textures should by ordered by drawing order
    // e.g. particle_texture will render first, then particle_texture2
```

```
    //      after some time
    "mymod:particle_texture2"
  ]
}
```

若要引用一个粒子纹理， `TextureSheetParticle` 的子类型应采用 `SpriteSet` 或从 `SpriteSet` 获得的 `TextureAtlasSprite` 。 `SpriteSet` 包含一个纹理列表，这些纹理引用了我们的 `ParticleDescription` 定义的sprite。 `SpriteSet` 有两个方法，这两个方法都以不同的方法获取 `TextureAtlasSprite` 。第一种方法接受两个整数。其背后的实现允许sprite在老化时进行纹理更改。第二种方法接受一个 `Random` 实例，从sprite集中获取随机纹理。可以使用 `SpriteSet` 中的一个辅助方法在 `TextureSheetParticle` 中设置sprite： `#pickSprite` 使用拾取纹理的随机方法， `#setSpriteFromAge` 使用两个整数的百分比方法拾取纹理。

要注册这些粒子纹理，需要向 `RegisterParticleProvidersEvent#registerSpriteSet` 方法提供一个 `SpriteParticleRegistration` 。此方法接收一个 `SpriteSet` ，其中包含粒子的相关sprite集，并创建一个 `ParticleProvider` 来创建粒子。最简单的实现方法可以通过在某个类上实现 `ParticleProvider` 并让构造函数接受 `SpriteSet` 来完成。然后， `SpriteSet` 可以正常地传递给粒子。

> **注意**
>
> 如果你注册的是仅包含一个纹理的 `TextureSheetParticle` 子类型，则可以转而向 `#registerSprite` 方法提供 `ParticleProvider$Sprite` ，其与 `ParticleProvider` 具有基本相同的功能接口方法。

9.1.2 生成一个粒子

粒子可以在任一存档实例中生成。但是，每一端都有一种特定的方式来生成粒子。如果在 `ClientLevel` 上，可以调用 `#addParticle` 来生成粒子，或者可以调用 `#addAlwaysVisibleParticle` 以生成从任何距离可见的粒子。如果在 `ServerLevel` 上，则可以调用 `#sendParticles` 向客户端发送数据包以生成粒子。在服务端上调用两个 `ClientLevel` 方法将会一无所获。

9.2 音效

9.2.1 术语

术语	描述
音效事件	触发音效效果的东西。例子包括 `minecraft:block.anvil.hit` 或 `botania:spreader_fire` 。
音效类别	音效的类别，例如 `player` 、 `block` 或只不过是 `master` 。音效设置GUI中的滑块展示这些类别。
音效文件	字面意义上的磁盘上播放的文件：一个.ogg文件。

9.2.2 sounds.json

此JSON定义音效事件，并定义它们播放的音效文件、字幕等。音效事件用 `ResourceLocation` 标识。 `sounds.json` 应该位于资源命名空间的根目录（ `assets/<namespace>/sounds.json` ），且在该命名空间中定义音效事件（ `assets/<namespace>/soundes.json` 在名称空间 `namespace` 中定义音效事件。）。

原版wiki上提供了完整的规范，但这个例子强调了重要的部分：

```
{
  "open_chest": {
    "subtitle": "mymod.subtitle.open_chest",
    "sounds": [ "mymod:open_chest_sound_file" ]
  },
  "epic_music": {
    "sounds": [
      {
        "name": "mymod:music/epic_music",
        "stream": true
      }
    ]
  }
}
```

在顶级对象的下面，每个键都对应一个音效事件。请注意，没有给出命名空间，因为它取自JSON本身的命名空间。每个事件指定启用字幕时要显示的本地化翻译键。最后，指定要播放的实际音效文件。请注意，该值是一个数组；如果指定了多个音效文件，则每当触发音效事件时，游戏将随机选择一个播放。

这两个示例代表了指定音效文件的两种不同方式。wiki有精确的细节，但一般来说，长音效文件（如背景音乐或音乐光盘）应该使用第二种形式，因为"stream"参数告诉Minecraft不要将整个音效文件加载到内存中，而是从磁盘流形式传输。第二种形式还可以指定音效文件的音量、音高和重量。

在所有情况下，命名空间 `namespace` 和路径 `path` 的音效文件路径都是 `assets/<namespace>/sounds/<path>.ogg` 。因此， `mymod:open_chest_sound_file` 指向 `assets/mymod/sounds/open_chest_sound_file.ogg` ， 而 `mymod:music/epic_music` 指向 `assets/mymod/sounds/music/epic_music.ogg` 。

`sounds.json` 可以是数据生成的。

9.2.3 创建音效事件

为了引用服务端上的音效，必须创建一个在 `sounds.json` 中包含相应条目的 `SoundEvent` 。然后必须对 `SoundEvent` 进行注册。通常，用于创建音效事件的位置应设置为其注册表名称。

`SoundEvent` 作为对音效的一个引用，并被传递以播放它们。如果一个模组有API，应该在API中公开它的 `SoundEvent` 。

> **注意**
>
> 只要音效在 `sounds.json` 中被注册，它就仍然可以在逻辑客户端上被引用，而不管是否存在引用其的 `SoundEvent` 。

9.2.4 播放音效

原版有很多播放音效的方法，有时很难清楚该用哪种。

请注意，每个方法都要接受一个 `SoundEvent` ，即上面注册的事件。此外，术语 “服务端行为” 和 “客户端行为” 指其分别的[逻辑端][side]。

`Level`

1. `playSound(Player, BlockPos, SoundEvent, SoundSource, volume, pitch)`
   - 简单地转发到重载 (2)，在给定的 `BlockPos` 的每个坐标上加0.5。
2. `playSound(Player, double x, double y, double z, SoundEvent, SoundSource, volume, pitch)`
   - 客户端行为：如果传入的玩家是客户端玩家，则向客户端玩家播放该音效事件。
   - 服务端行为：向附近的所有人播放音效事件，除了传入的玩家以外。玩家可以为 `null` 。
   - 用法： 行为之间的对应关系意味着这两个方法将从一些玩家启动的代码中调用，这些代码将同时在两逻辑端运行：逻辑客户端处理向用户播放，逻辑服务端处理其他所有听到它的人，而不向原始用户重新播放。它们还可以用于在服务端端的任何位置播放任何音效，方法是在逻辑服务端上调用它并传入 `null` 玩家，从而让每个人都能听到。
3. `playLocalSound(double x, double y, double z, SoundEvent, SoundSource, volume, pitch, distanceDelay)`
   - 客户端行为：只是在客户端存档播放音效事件。如果 `distanceDelay` 为 `true` ，则根据音效与玩家的距离来延迟音效。
   - 服务端行为：不做任何事情。
   - 用法： 此方法仅适用于客户端，因此对于在自定义数据包中发送的音效或其他仅客户端效果类型的音效非常有用。打雷就用了该方法。

`ClientLevel`

1. `playLocalSound(BlockPos, SoundEvent, SoundSource, volume, pitch, distanceDelay)`
   - 简单地转发到 `Level` 的overload (3)，在给定的 `BlockPos` 的每个坐标上加0.5。

`Entity`

1. `playSound(SoundEvent, volume, pitch)`
   - 简单地转发到 `Level` 的overload (2)，将玩家传递为 `null` 。
   - 客户端行为：不做任何事情。
   - 服务端行为：向该实体所在位置的所有人播放音效事件。
   - 用法：在服务端从任何非玩家实体发出任何音效。

`Player`

1. `playSound(SoundEvent, volume, pitch)` (overriding the one in `Entity`)
   - 简单地转发到 `Level` 的overload (2)，将玩家传递为 `null` 。
   - 客户端行为：不做任何事情，参见 `LocalPlayer` 中的重载。
   - 服务端行为：向附近除了该玩家以外的所有人播放该音效。
   - 用法：参见 `LocalPlayer` 。

`LocalPlayer`

1. `playSound(SoundEvent, volume, pitch)` (overriding the one in `Player`)
   - 简单地转发到 `Level` 的overload (2)，将玩家传递为 `this` 。
   - 客户端行为：仅仅播放该音效事件。
   - 服务端行为：该方法仅客户端适用。
   - 用法：就像 `Level` 中的方法一样，玩家类中的这两个重写似乎是针对在两端同时运行的代码。客户端处理向用户播放音效，而服务端处理其他所有听到音效的人，而不向原始用户重新播放。

10. 数据储存

10.1 Capability系统

Capability允许以动态和灵活的方式公开Capability，而不必直接实现许多接口。

一般来说，每个Capability都以接口的形式提供了一个Capability。

Forge为BlockEntity、Entity、ItemStack、Level和LevelChunk添加了Capability支持，这些Capability可以通过事件附加它们，也可以通过重写你自己的对象实现中的Capability方法来公开。这将在接下来的章节中进行更详细的解释。

10.1.1 Forge提供的Capability

Forge提供三种Capability：`IItemHandler`、`IFluidHandler`和`IEnergyStorage`。

`IItemHandler`公开了一个用于处理物品栏Slot的接口。它可以应用于BlockEntity（箱子、机器等）、Entity（额外的玩家Slot、生物/生物物品栏/袋子）或ItemStack（便携式背包等）。它用一个自动化友好的系统取代了旧的`Container`和`WorldlyContainer`。

`IFluidHandler`公开了一个用于处理流体物品栏的接口。它也可以应用于BlockEntitiy、Entity或ItemStack。

`IEnergyStorage`公开了一个用于处理能源容器的接口。它可以应用于BlockEntity、Entity或ItemStack。它基于TeamCoFH的RedstoneFlux API。

10.1.2 使用现存的Capability

如前所述，BlockEntity、Entity和ItemStack通过`ICapabilityProvider`接口实现了Capability提供者Capability。此接口添加了方法`#getCapability`，该方法可用于查询相关提供者对象中存在的Capability。

为了获得一个Capability，你需要通过它的唯一实例来引用它。在`IItemHandler`的情况下，此Capability主要存储在`ForgeCapabilities#ITEM_HANDLER`中，但也可以使用`CapabilityManager#get`获取其他实例引用。

```
public static final Capability<IItemHandler> ITEM_HANDLER = CapabilityManager.get(new CapabilityToken<>(){});
```

当被调用时，`CapabilityManager#get`为你的相关类型提供一个非null的Capability。匿名的`CapabilityToken`允许Forge保持软依赖系统，同时仍然拥有获得正确Capability所需的泛型信息。

重要

即使你在任何时候都可以使用非null的Capability，但这并不意味着该Capability本身是可用的或已注册的。这可以通过`Capability#isRegistered`进行检查。

`#getCapability`方法有另一个参数，类型为`Direction`，可用于请求那一面的特定实例。如果传递`null`，则可以假设请求来自方块内，或者来自某个侧面没有意义的地方，例如不同的维度。在这种情况下，将请求一个不关侧面的一个通用的Capability实例。`#getCapability`的返回类型将对应于传递给方法的Capability中声明的类型的`LazyOptional`。对于物品处理器Capability，其为`LazyOptional<IItemHandler>`。如果该Capability不适用于特定的提供者，它将返回一个空的`LazyOptional`。

10.1.3 公开一个Capability

为了公开一个Capability，你首先需要一个底层Capability类型的实例。请注意，你应该为每个保有该Capability的对象分配一个单独的实例，因为该Capability很可能与所包含的对象绑定。

在 `IItemHandler` 的情况下，默认实现使用 `ItemStackHandler` 类来指定多个Slot，该类在构造函数中有一个可选参数。然而，应避免依赖这些默认实现的存在，因为Capability系统的目的是防止在不存在Capability的情况下出现加载错误，因此如果Capability已注册，则应在检查测试之后对实例化进行保护（请参阅上一节中关于 `CapabilityManager#get` 的备注）。

一旦你拥有了自己的Capability接口实例，你将希望通知Capability系统的用户你公开了此Capability，并提供接口引用的 `LazyOptional` 。这是通过重写 `#getCapability` 方法来完成的，并将Capability实例与你要公开的Capability进行比较。如果你的机器根据被查询的一侧有不同的Slot，你可以使用 `side` 参数进行测试。对于实体和物品栈，此参数可以忽略，但仍然可以将侧面作为上下文，例如玩家上的不同护甲Slot（ `Direction#UP` 暴露玩家的头盔Slot），或物品栏中的周围方块（ `Direction#WEST` 暴露熔炉的输入Slot）。不要忘记回到 `super` ，否则现有的附加Capability将停止工作。

在提供者生命周期结束时，必须通过 `LazyOptional#invalidate` 使Capability失效。对于拥有的BlockEntitiy和Entity，`LazyOptional` 可以在 `#invalidateCaps` 内失效。对于非拥有者提供者，提供失效过程的Runnable应传递到 `AttachCapabilitiesEvent#addListener` 中。

```
// 在你BlockEntity子类中的某处
LazyOptional<IItemHandler> inventoryHandlerLazyOptional;

// 被提供的对象（例如：() -> inventoryHandler）
// 确保惰性，因为初始化只应在需要时发生
inventoryHandlerLazyOptional = LazyOptional.of(inventoryHandlerSupplier);

@Override
public <T> LazyOptional<T> getCapability(Capability<T> cap, Direction side) {
  if (cap == ForgeCapabilities.ITEM_HANDLER) {
    return inventoryHandlerLazyOptional.cast();
  }
  return super.getCapability(cap, side);
}

@Override
public void invalidateCaps() {
  super.invalidateCaps();
  inventoryHandlerLazyOptional.invalidate();
}
```

提示

如果给定对象上只公开了一个Capability，则可以使用 `Capability#orEmpty` 作为if/else语句的替代语句。

```
@Override
public <T> LazyOptional<T> getCapability(Capability<T> cap, Direction side) {
  return ForgeCapabilities.ITEM_HANDLER.orEmpty(cap, inventoryHandlerLazyOptional);
}
```

`Item` 是一种特殊情况，因为它们的Capability提供者存储在 `ItemStack` 上。相反的是，应该通过 `Item#initCapabilities` 附加提供者。其应该在物品栈的生命周期中保持你的Capability。

强烈建议在代码中使用直接检查来测试Capability，而不是试图依赖Map或其他数据结构，因为每个游戏刻都可以由许多对象进行Capability测试，并且它们需要尽可能快，以避免减慢游戏速度。

10.1.4 Capability的附加

如前所述，可以使用 `AttachCapabilitiesEvent` 将Capability附加到现有提供者、 `Level` 和 `LevelChunk` 。同一事件用于所有可以提供Capability的对象。 `AttachCapabilitiesEvent` 有5个有效的泛型类型，提供以下事件：

- `AttachCapabilitiesEvent<Entity>` ：仅为实体触发。
- `AttachCapabilitiesEvent<BlockEntity>` ：仅为方块实体触发。
- `AttachCapabilitiesEvent<ItemStack>` ：仅为物品栈触发。
- `AttachCapabilitiesEvent<Level>` ：仅为存档触发。
- `AttachCapabilitiesEvent<LevelChunk>` ：仅为存档区块触发。

泛型类型不能比上述类型更具体。例如：如果要将Capability附加到 `Player` ，则必须订阅 `AttachCapabilitiesEvent<Entity>` ，然后在附加Capability之前确定所提供的对象是 `Player` 。

在所有情况下，该事件都有一个方法 `#addCapability` ，可用于将Capability附加到目标对象。不是将Capability本身添加到列表中，而是添加Capability提供者，这些提供者有机会仅从某些面返回Capability。虽然提供者只需要实现 `ICapabilityProvider` ，但如果该Capability需要持久存储数据，则可以实现 `ICapabilitySerializable<T extends Tag>` ，该Capability除了返回Capability外，还将提供标签保存/加载Capability。

有关如何实现 `ICapabilityProvider` 的信息，请参阅公开一个Capability部分。

10.1.5 创建你自己的Capability

Capability可通过以下两种方式之一被注册： `RegisterCapabilitiesEvent` 或 `@AutoRegisterCapability` 。

RegisterCapabilitiesEvent

通过向 `#register` 方法提供Capability类型的类，可以使用 `RegisterCapabilitiesEvent` 注册Capability。该事件在模组事件总线上被处理。

```
@SubscribeEvent
public void registerCaps(RegisterCapabilitiesEvent event) {
  event.register(IExampleCapability.class);
}
```

@AutoRegisterCapability

Capability也可通过使用 `@AutoRegisterCapability` 注释以被注册。

```
@AutoRegisterCapability
public interface IExampleCapability {
  // ...
}
```

10.1.6 LevelChunk和BlockEntity的Capability的持久化

与Level、Entity和ItemStack不同，LevelChunk和BlockEntity只有在标记为脏时才会写入磁盘。因此，LevelChunk或BlockEntity具有持久状态的Capability实现应确保无论何时其状态发生变化，其所有者都被标记为脏。

`ItemStackHandler` 通常用于BlockEntity中的物品栏，它有一个可重写的方法 `void onContentsChanged(int slot)` ，用于将BlockEntity标记为脏。

```
public class MyBlockEntity extends BlockEntity {

  private final IItemHandler inventory = new ItemStackHandler(...) {
    @Override
```

```
    protected void onContentsChanged(int slot) {
      super.onContentsChanged(slot);
      setChanged();
    }
  }

  // ...
}
```

10.1.7 向客户端同步数据

默认情况下，Capability数据不会发送到客户端。为了改变这一点，模组必须使用数据包管理自己的同步代码。

在三种不同的情况下，你可能希望发送同步数据包，所有这些情况都是可选的：

1. 当实体在存档中生成或放置方块时，你可能希望与客户端共享初始化指定的值。
2. 当存储的数据发生更改时，你可能需要通知部分或全部正在监视的客户端。
3. 当新客户端开始查看实体或方块时，你可能希望将现有数据通知它。

有关实现网络数据包的更多信息，请参阅网络页面。

10.1.8 在玩家死亡时的持久化

默认情况下，Capability数据不会在死亡时持续存在。为了改变这一点，在重生过程中克隆玩家实体时，必须手动复制数据。

这可以通过 `PlayerEvent$Clone` 完成，方法是从原始实体读取数据并将其分配给新实体。在这种情况下， `#isWasDeath` 方法可以用于区分死后重生和从末地返回。这一点很重要，因为从末地返回时数据已经存在，因此在这种情况下必须注意不要重复值。

10.1.9 从IExtendedEntityProperty迁移

尽管Capability系统可以完成IEEP（IExtendedEntityProperty）所做的一切，甚至更多，但这两个概念并不完全匹配。本节将解释如何将现有IEEP转换为Capability。

这是IEEP概念及其等效Capability概念的快速列表：

- 属性名称/id (`String`)：Capability 键 (`ResourceLocation`)
- 注册 (`EntityConstructing`)：附加 (`AttachCapabilitiesEvent<Entity>`)， `Capability` 的真正的注册发生于 `FMLCommonSetupEvent` 期间。
- 标签读/写方法：不会自动发生。在该事件中附加一个 `ICapabilitySerializable` 并运行 `serializeNBT` / `deserializeNBT` 作为读/写方法。

快速转换指南：

1. 将IEEP键/id字符串转换为 `ResourceLocation` （其将使用你的MODID作为命名空间）。
2. 在你的处理器类中（不是实现你的Capability接口的类），创建一个容纳Capability实例的字段。
3. 将 `EntityConstructing` 事件转换为 `AttachCapabilitiesEvent` ，且与查询IEEP不同，你将想要附加一个 `ICapabilityProvider` （可能为 `ICapabilitySerializable` ，其允许保存到一个标签/从一个标签加载）。
4. 创建一个注册方法，如果你没有的话（你可能在注册你的IEEP事件处理器的地方有一个）并在该方法中运行Capability注册函数。

10.2 Saved Data

Saved Data（SD）系统是存档Capability功能的替代方案，可以按存档附加数据。

10.2.1 声明

Each SD implementation must subtype the `SavedData` class. There are two important methods to be aware of: 每个SD实现都必须继承 `SavedData` 类。有两种重要方法需要注意：

- `save`：允许实现将NBT数据写入该存档。
- `setDirty`：在更改数据后必须调用的方法，以通知游戏有需要写入的更改。如果未调用，将不会调用 `#save`，并且现有数据将持久存在。

10.2.2 附加到存档

任何 `SavedData` 都是动态加载和/或附加到一个存档的。因此，如果一个 `SavedData` 从来没有在一个存档上创建过，那么它就不存在了。

`SavedData` 是从 `DimensionDataStorage` 创建和加载的，借助 `ServerChunkCache#getDataStorage` 或 `ServerLevel#getDataStorage` 都可以访问该存储。从那里，您可以通过调用 `DimensionDataStorage#computeIfAbsent` 来获取或创建SD的实例。这将尝试获取SD的当前实例（如果存在），或者创建一个新实例并加载所有可用数据。

`DimensionDataStorage#computeIfAbsent` 接受三个参数：一个将NBT数据加载到SD并返回它的函数，一个构造SD新实例的Supplier，以及存储在所实现的存档的 `data` 文件夹中的 `.dat` 文件的名称。

例如，如果一个SD在下界中被命名为"example"，那么一个文件将在 `./<level_folder>/DIM-1/data/example.dat` 创建并且将这样实现：

```
// 在某个类中
public ExampleSavedData create() {
  return new ExampleSavedData();
}

public ExampleSavedData load(CompoundTag tag) {
  ExampleSavedData data = this.create();
  // 加载saved data
  return data;
}

// 在该类的某个方法中
netherDataStorage.computeIfAbsent(this::load, this::create, "example");
```

要在多个存档之间保持SD，应将SD连接到主世界，其可以从 `MinecraftServer#overworld` 获得。主世界是唯一一个从未完全卸载的维度，因此非常适合在其上存储多存档数据。

10.3 编解码器（Codecs）

编解码器（Codecs）是源于Mojang的DataFixerUpper的一个序列化工具，用于描述对象如何在不同格式之间转换，例如JSON的 `JsonElement` 和NBT的 `Tag` 。

10.3.1 编解码器的使用

编解码器主要用于将Java对象编码或序列化为某种数据格式类型，并将格式化的数据对象解码或反序列化为其关联的Java类型。这通常分别使用 `Codec#encodeStart` 和 `Codec#parse` 来完成。

DynamicOps

为了确定要编码和解码的中间文件格式， `#encodeStart` 和 `#parse` 都需要一个 `DynamicOps` 实例来定义该格式中的数据。

DataFixerUpper库包含 `JsonOps` ，用于对存储在 `Gson` 的 `JsonElement` 实例中的JSON数据进行编码。 `JsonOps` 支持两个版本的 `JsonElement` 序列化：定义标准JSON文件的 `JsonOps#INSTANCE` 和允许将数据压缩为单个字符串的 `JsonOps#COMPRESSED` 。

```
// 让exampleCodec代表一个Codec<ExampleJavaObject>
// 让exampleObject是一个ExampleJavaObject
// 让exampleJson是一个JsonElement

// 将Java对象编码为常规的JsonElement
exampleCodec.encodeStart(JsonOps.INSTANCE, exampleObject);

// 将Java对象编码为压缩的JsonElement
exampleCodec.encodeStart(JsonOps.COMPRESSED, exampleObject);

// 将JsonElement解码为Java对象
// 假设JsonElement被普通地转换
exampleCodec.parse(JsonOps.INSTANCE, exampleJson);
```

Minecraft还提供了 `NbtOps` 来对存储在 `Tag` 实例中的NBT数据进行编解码。其可以使用 `NbtOps#INSTANCE` 被引用。

```
// 让exampleCodec代表一个Codec<ExampleJavaObject>
// 让exampleObject是一个ExampleJavaObject
// 让exampleNbt是一个Tag

// 将Java对象编码为Tag
exampleCodec.encodeStart(JsonOps.INSTANCE, exampleObject);

// 将Tag解码为Java对象
exampleCodec.parse(JsonOps.INSTANCE, exampleNbt);
```

格式的转换

`DynamicOps` 还可以单独用于在两种不同的编码格式之间进行转换。这可以使用 `#convertTo` 并提供 `DynamicOps` 格式和要转换的编码对象来完成。

```
// 将Tag转换为JsonElement
// 让exampleTag是一个Tag
JsonElement convertedJson = NbtOps.INSTANCE.convertTo(JsonOps.INSTANCE, exampleTag);
```

DataResult

使用编解码器编码或解码的数据返回一个 `DataResult`，它保存转换后的实例或一些错误数据，具体取决于转换是否成功。转换成功后，`#result` 提供的 `Optional` 将包含成功转换的对象。如果转换失败，`#error` 提供的 `Optional` 将包含 `PartialResult`，其中包含错误消息和部分转换的对象，具体取决于编解码器。

此外，`DataResult` 上有许多方法可用于将结果或错误转换为所需格式。例如，`#resultOrPartial` 将返回一个 `Optional`，其中包含成功时的结果，以及失败时部分转换的对象。该方法接收字符串Consumer，以确定如何报告错误消息（如果存在）。

```
// 让exampleCodec代表一个Codec<ExampleJavaObject>
// 让exampleJson是一个JsonElement

// 将JsonElement解码为Java对象
DataResult<ExampleJavaObject> result = exampleCodec.parse(JsonOps.INSTANCE, exampleJson);

result
  // 获取结果或部分结果（当错误时），并报告错误消息
  .resultOrPartial(errorMessage -> /* 处理错误消息 */)
  // 如果结果或部分结果存在，做一些事情
  .ifPresent(decodedObject -> /* 处理解码后的对象 */);
```

10.3.2 现存的编解码器

原始类型

`Codec` 类包含某些已定义的原始类型的编解码器的静态实例。

Codec	Java类型
`BOOL`	`Boolean`
`BYTE`	`Byte`
`SHORT`	`Short`
`INT`	`Integer`
`LONG`	`Long`
`FLOAT`	`Float`
`DOUBLE`	`Double`
`STRING`	`String`
`BYTE_BUFFER`	`ByteBuffer`
`INT_STREAM`	`IntStream`
`LONG_STREAM`	`LongStream`
`PASSTHROUGH`	`Dynamic<?>` *
`EMPTY`	`Unit` **

* `Dynamic` 是一个对象，它包含以支持的 `DynamicOps` 格式编码的值。这些通常用于将编码对象格式转换为其他编码对象格式。

** `Unit` 是一个用于表示 `null` 对象的对象。

原版和**Forge**

Minecraft和Forge为经常编码和解码的对象定义了许多编解码器。一些示例包括 `ResourceLocation` 的 `ResourceLocation#CODEC` ， `DateTimeFormatter#ISO_INSTANT` 格式的 `Instant` 的 `ExtraCodecs#INSTANT_ISO8601` ，以及 `CompoundTag` 的 `CompoundTag#CODEC` 。

> **警告**
>
> `CompoundTag` 无法使用 `JsonOps` 解码JSON中的数字列表。转换时， `JsonOps` 将数字设置为其最窄的类型。 `ListTag` 强制为其数据指定一个特定类型，因此具有不同类型的数字（例如， `64` 将是 `byte` ， `384` 为 `short` ）将在转换时引发错误。

原版和Forge注册表也具有注册表所包含对象类型的编解码器（例如 `Registry#BLOCK` 或 `ForgeRegistries#BLOCKS` 具有 `Codec<Block>` ）。 `Registry#byNameCodec` 和 `IForgeRegistry#getCodec` 将把注册表对象编码为其注册表名称，或者如果被压缩，则编码为整数标识符。原版注册表还有一个 `Registry#holderByNameCodec` ，它编码为注册表名称，并解码为 `Holder` 中包装的注册表对象。

10.3.3 创建编解码器

可以创建用于对任何对象进行编码和解码的编解码器。为了便于理解，将展示等效的编码JSON。

记录

编解码器可以通过使用记录来定义对象。每个记录编解码器都定义了具有显式命名字段的任何对象。创建记录编解码器的方法有很多，但最简单的是通过 `RecordCodecBuilder#create` 。

`RecordCodecBuilder#create` takes in a function which defines an `Instance` and returns an application (`App`) of the object. A correlation can be drawn to creating a class *instance* and the constructors used to *apply* the class to the constructed object. `RecordCodecBuilder#create` 接受一个定义 `Instance` 的函数，并返回对象的应用（ `App` ）。一个为创建类实例和用于将该类应用于所构造对象的构造函数的关联可被绘制。

```
// 要为其创建编解码器的某个对象
public class SomeObject {

  public SomeObject(String s, int i, boolean b) { /* ... */ }

  public String s() { /* ... */ }

  public int i() { /* ... */ }

  public boolean b() { /* ... */ }
}
```

字段

一个 `Instance` 可以使用 `#group` 定义多达16个字段。每个字段都必须是一个应用，定义为其创建对象的实例和对象的类型。满足这一要求的最简单方法是使用 `Codec` ，设置要解码的字段的名称，并设置用于编码字段的getter。

如果字段是必需的，则可以使用 `#fieldOf` 从 `Codec` 创建字段；如果字段被包装在 `Optional` 或默认值中，则使用 `#optionalFieldOf` 创建字段。任一方法都需要一个字符串，该字符串包含编码对象中字段的名称。然后，可以使用 `#forGetter` 设置用于对字段进行编码的getter，接受一个给定对象并返回字段数据的函数。

从那里，可以通过 `#apply` 应用生成的产品，以定义实例应如何构造应用的对象。为了方便起见，分组字段应该按照它们在构造函数中出现的顺序列出，这样函数就可以简单地作为构造函数方法引用。

```
public static final Codec<SomeObject> RECORD_CODEC = RecordCodecBuilder.create(instance -> // 给定一个实例
  instance.group( // 定义该实例内的字段
    Codec.STRING.fieldOf("s").forGetter(SomeObject::s), // 字符串
    Codec.INT.optionalFieldOf("i", 0).forGetter(SomeObject::i), // 整数，当字段不存在时默认为0
    Codec.BOOL.fieldOf("b").forGetter(SomeObject::b) // 布尔值
  ).apply(instance, SomeObject::new) // 定义如何创建该对象
);
```

```
// 已编码的SomeObject
{
  "s": "value",
  "i": 5,
  "b": false
}

// 另一个已编码的SomeObject
{
  "s": "value2",
  // i被忽略，默认为0
  "b": true
}
```

转换器

编解码器可以通过映射方法转换为等效或部分等效的表示。每个映射方法都有两个函数：一个将当前类型转换为新类型，另一个将新类型转换回当前类型。这是通过 `#xmap` 函数完成的。

```
// A类
public class ClassA {

  public ClassB toB() { /* ... */ }
}

// 另一个等效的类
public class ClassB {

  public ClassA toA() { /* ... */ }
}

// 假设有一个编解码器A_CODEC
public static final Codec<ClassB> B_CODEC = A_CODEC.xmap(ClassA::toB, ClassB::toA);
```

如果一个类型是部分等效的，这意味着在转换过程中存在一些限制，则存在返回 `DataResult` 的映射函数，每当达到异常或无效状态时，该函数可用于返回错误状态。

A是否完全等效于B	B是否完全等效于A	转换方法
是	是	`#xmap`
是	否	`#flatComapMap`
否	是	`#comapFlatMap`
否	否	`#flatXMap`

```
// 给定一个字符串编码器用于转换为一个整数
// 并非所有字符串都能成为整数（A不完全等效于B）
// 所有整数都能成为字符串（B完全等效于A）
public static final Codec<Integer> INT_CODEC = Codec.STRING.comapFlatMap(
  s -> { // 返回含有错误或失败的数据结果
    try {
      return DataResult.success(Integer.valueOf(s));
    } catch (NumberFormatException e) {
      return DataResult.error(s + " is not an integer.");
    }
  },
  Integer::toString // 常规函数
);
```

```
// 将会返回5
"5"

// 将会产生错误，不是一个整数
"value"
```

范围编解码器

范围编解码器是 `#flatXMap` 的实现，如果值不包含在设置的最小值和最大值之间，则返回错误 `DataResult` 。如果超出界限，该值仍将作为部分结果提供。分别通过 `#intRange` 、 `#floatRange` 和 `#doubleRange` 实现了整数（int）、浮点数（float）和双精度小数（double）。

```
public static final Codec<Integer> RANGE_CODEC = Codec.intRange(0, 4);
```

```
// 将会合法，在[0, 4]范围内
4

// 将会产生错误，在[0, 4]范围外
5
```

默认值

如果编码或解码的结果失败，则可以通过 `Codec#orElse` 或 `Codec#orElseGet` 提供默认值。

```
public static final Codec<Integer> DEFAULT_CODEC = Codec.INT.orElse(0); // Can also be a supplied value via #orElseGet
```

```
// 不是一个整数，默认为0
"value"
```

Unit

提供代码内的值并编码为空的编解码器可以使用 `Codec#unit` 来表示。如果编解码器在数据对象中使用了不可编码的条目，这将非常有用。

```
public static final Codec<IForgeRegistry<Block>> UNIT_CODEC = Codec.unit(
  () -> ForgeRegistries.BLOCKS // 也可以是一个原始值
);
```

```
// 此处无内容，将会返回方块注册表编解码器
```

List

对象列表的编解码器可以通过 `Codec#listOf` 从对象编解码器生成。

```
// BlockPos#CODEC是一个Codec<BlockPos>
public static final Codec<List<BlockPos>> LIST_CODEC = BlockPos.CODEC.listOf();
```

```
// 已编码的List<BlockPos>
[
  [1, 2, 3], // BlockPos(1, 2, 3)
  [4, 5, 6], // BlockPos(4, 5, 6)
  [7, 8, 9]  // BlockPos(7, 8, 9)
]
```

使用列表编解码器解码的列表对象存储在不可变列表中。如果需要可变列表，则应将转换器应用于列表编解码器。

Map

键和值对象映射（Map）的编解码器可以通过 `Codec#unboundedMap` 从两个编解码器生成。无边界映射可以指定任何基于字符串或经过字符串转换的值作为键。

```
// BlockPos#CODEC是一个Codec<BlockPos>
public static final Codec<Map<String, BlockPos>> MAP_CODEC = Codec.unboundedMap(Codec.STRING, BlockPos.CODEC);
```

```
// 已编码的Map<String, BlockPos>
{
  "key1": [1, 2, 3], // key1 -> BlockPos(1, 2, 3)
  "key2": [4, 5, 6], // key2 -> BlockPos(4, 5, 6)
  "key3": [7, 8, 9]  // key3 -> BlockPos(7, 8, 9)
}
```

使用无界映射编解码器解码的映射对象存储在不可变映射中。如果需要一个可变映射，则应该将转换器应用于映射编解码器。

警告

无界映射仅支持对字符串进行编码/解码的键。键值对列表编解码器可以用来绕过这个限制。

Pair

对象对的编解码器可以通过 `Codec#pair` 从两个编解码器生成。

成对编解码器通过首先解码成对中的左对象，然后取编码对象的剩余部分并从中解码右对象来解码对象。因此，编解码器必须在解码后表达关于编码对象的某些内容（例如记录），或者必须将它们扩充为 `MapCodec`，并通过 `#codec` 转换为常规编解码器。这通常可以通过使编解码器成为某个对象的字段来实现。

```
public static final Codec<Pair<Integer, String>> PAIR_CODEC = Codec.pair(
  Codec.INT.fieldOf("left").codec(),
  Codec.STRING.fieldOf("right").codec()
);
```

```
// 已编码的Pair<Integer, String>
{
  "left": 5,       // fieldOf查询'left'键以获取左对象
  "right": "value" // fieldOf查询'right'键以获取右对象
}
```

提示

可以使用转换器应用的键值对列表对具有非字符串键的映射编解码器进行编码/解码。

Either

用于编码/解码某些对象数据的两种不同方法的编解码器可以通过 `Codec#either` 从两个编解码器生成。

Either编解码器尝试使用第一编解码器对对象进行解码。如果失败，它将尝试使用第二个编解码器进行解码。如果也失败了，那么 `DataResult` 将只包含第二个编解码器失败的错误。

```
public static final Codec<Either<Integer, String>> EITHER_CODEC = Codec.either(
  Codec.INT,
  Codec.STRING
);
```

```
// 已编码的Either$Left<Integer, String>
5

// 已编码的Either$Right<Integer, String>
"value"
```

提示

这可以与转换器结合使用，从两种不同的编码方法中获取特定对象。

Dispatch

编解码器可以具有子解码器，子解码器可以通过 `Codec#dispatch` 基于某个指定类型对特定对象进行解码。这通常用于包含编解码器的注册表中，例如规则测试或方块放置器。

Dispatch编解码器首先尝试从某个字符串关键字（通常为 `type` ）中获取编码类型。从那里，类型被解码，为用于解码实际对象的特定编解码器调用getter。如果用于解码对象的 `DynamicOps` 压缩了其映射，或者对象编解码器本身没有扩充为 `MapCodec` （例如记录或已部署的基本类型），则需要将对象存储在 `value` 键中。否则，对象将在与其余数据相同的级别上进行解码。

```
// 定义我们的对象
public abstract class ExampleObject {

  // 定义用于指定要编码的对象类型的方法
  public abstract Codec<? extends ExampleObject> type();
}

// 创建存储字符串的简单对象
public class StringObject extends ExampleObject {

  public StringObject(String s) { /* ... */ }

  public String s() { /* ... */ }

  public Codec<? extends ExampleObject> type() {
    // 一个已注册的注册表对象
    // "string":
    //   Codec.STRING.xmap(StringObject::new, StringObject::s)
    return STRING_OBJECT_CODEC.get();
  }
}

// 创建存储字符串和整数的复杂对象
public class ComplexObject extends ExampleObject {

  public ComplexObject(String s, int i) { /* ... */ }

  public String s() { /* ... */ }

  public int i() { /* ... */ }

  public Codec<? extends ExampleObject> type() {
    // 一个已注册的注册表对象
    // "complex":
    //   RecordCodecBuilder.create(instance ->
    //     instance.group(
    //       Codec.STRING.fieldOf("s").forGetter(ComplexObject::s),
    //       Codec.INT.fieldOf("i").forGetter(ComplexObject::i)
    //     ).apply(instance, ComplexObject::new)
    //   )
    return COMPLEX_OBJECT_CODEC.get();
  }
}

// 假设有一个IForgeRegistry<Codec<? extends ExampleObject>> DISPATCH
public static final Codec<ExampleObject> = DISPATCH.getCodec() // 获取Codec<Codec<? extends ExampleObject>>
  .dispatch(
    ExampleObject::type, // 从特定对象获取编解码器
    Function.identity() // 从注册表获取编解码器
  );
```

```
// 简单对象
{
```

```
  "type": "string", // 对于StringObject
  "value": "value" // MapCodec不需要编解码器类型参数，需要字段
}

// 复杂对象
{
  "type": "complex", // 对于ComplexObject

  // MapCodec不需要编解码器类型参数，可被内联
  "s": "value",
  "i": 0
}
```

11. 图形用户界面

11.1 菜单（Menus）

菜单（Menus）是图形用户界面（GUI）的一种后端类型；它们处理与某些代表的数据持有者交互所涉及的逻辑。菜单本身不是数据持有者。它们是允许用户间接修改内部数据持有者状态的视图。因此，数据持有者不应直接耦合到任何菜单，而应传入数据引用以便调用和修改。

11.1.1 MenuType

菜单是动态创建和删除的，因此不是注册表对象。因此，另一种工厂对象被注册，以方便创建和引用菜单的类型。对于菜单，其为 `MenuType` 。

`MenuType` 必须被注册。

MenuSupplier

`MenuType` 是通过将 `MenuSupplier` 和 `FeatureFlagSet` 传递给其构造函数来创建的。 `MenuSupplier` 表示一个函数，该函数接收容器的id和查看菜单的玩家的物品栏，并返回一个新创建的 `AbstractContainerMenu` 。

```
// 对于某个类型为DeferredRegister<MenuType<?>>的REGISTER
public static final RegistryObject<MenuType<MyMenu>> MY_MENU = REGISTER.register("my_menu", () -> new MenuType(MyMenu::new, FeatureF

// 在MyMenu，一个AbstractContainerMenu的子类中
public MyMenu(int containerId, Inventory playerInv) {
  super(MY_MENU.get(), containerId);
  // ...
}
```

> **注意**
>
> 容器id对于单个玩家是唯一的。这意味着，两个不同玩家上的相同容器id将代表两个不同的菜单，即使他们正在查看相同的数据持有者。

`MenuSupplier` 通常负责在客户端上创建一个菜单，其中包含用于存储来自服务端数据持有者的同步信息并与之交互的伪数据引用。

IContainerFactory

如果需要有关客户端的其他信息（例如数据持有者在世界中的位置），则可以使用子类 `IContainerFactory` 。除了容器id和玩家物品栏之外，这还提供了一个 `FriendlyByteBuf` ，它可以存储从服务端发送的附加信息。 `MenuType` 可以通过 `IForgeMenuType#create` 使用 `IContainerFactory` 创建。

```
// 对于某个类型为DeferredRegister<MenuType<?>>的REGISTER
public static final RegistryObject<MenuType<MyMenuExtra>> MY_MENU_EXTRA = REGISTER.register("my_menu_extra", () -> IForgeMenuType.cr

// 在MyMenuExtra，一个AbstractContainerMenu的子类中
public MyMenuExtra(int containerId, Inventory playerInv, FriendlyByteBuf extraData) {
  super(MY_MENU_EXTRA.get(), containerId);
  // 从buffer中存储附加信息
```

```
  // ...
}
```

11.1.2 AbstractContainerMenu

所有菜单都是从 `AbstractContainerMenu` 继承而来的。菜单包含两个参数，即表示菜单本身类型的 `MenuType` 和表示当前访问者的菜单唯一标识符的容器id。

> **重要**
>
> 玩家一次只能打开100个唯一的菜单。

每个菜单应该包含两个构造函数：一个用于初始化服务端上的菜单，另一个用于启动客户端上的菜单。用于初始化客户端菜单的构造函数是提供给 `MenuType` 的构造函数。服务端菜单构造函数包含的任何字段都应该具有客户端菜单构造函数的一些默认值。

```
// 客户端菜单构造函数
public MyMenu(int containerId, Inventory playerInventory) {
  this(containerId, playerInventory);
}

// 服务端菜单构造函数
public MyMenu(int containerId, Inventory playerInventory) {
  // ...
}
```

每个菜单实现必须实现两个方法： `#stillValid` 和 `#quickMoveStack` 。

`#stillValid` 和 `ContainerLevelAccess`

`#stillValid` 确定菜单是否应该为给定的玩家保持打开状态。这通常指向静态的 `#stillValid` ，它接受一个 `ContainerLevelAccess` 、该玩家和该菜单所附的 `Block` 。客户端菜单必须始终为该方法返回 `true` ，而静态的 `#stillValid` 默认为该方法。该实现检查玩家是否在数据存储对象所在的八个方块内。

`ContainerLevelAccess` 提供封闭范围内方块的当前存档和位置。在服务端上构建菜单时，可以通过调用 `ContainerLevelAccess#create` 创建新的访问。客户端菜单构造函数可以传入 `ContainerLevelAccess#NULL` ，这将不起任何作用。

```
// 客户端菜单构造函数
public MyMenuAccess(int containerId, Inventory playerInventory) {
  this(containerId, playerInventory, ContainerLevelAccess.NULL);
}

// 服务端菜单构造函数
public MyMenuAccess(int containerId, Inventory playerInventory, ContainerLevelAccess access) {
  // ...
}

// 假设该菜单已绑定到RegistryObject<Block> MY_BLOCK
@Override
public boolean stillValid(Player player) {
  return AbstractContainerMenu.stillValid(this.access, player, MY_BLOCK.get());
}
```

数据的同步

一些数据需要同时出现在服务端和客户端上才能显示给玩家。为此，菜单实现了数据同步的基本层，以便在当前数据与上次同步到客户端的数据不匹配时进行同步。对于玩家来说，这是每个tick都会检查的。

Minecraft默认支持两种形式的数据同步：通过 `Slot` 进行的 `ItemStack` 同步和通过 `DataSlot` 进行的整数同步。 `Slot` 和 `DataSlot` 是保存对数据存储的引用的视图，假设操作有效，玩家可以在屏幕中修改这些数据存储。这些可以通过 `#addSlot` 和 `#addDataSlot` 在菜单的构造函数中添加。

注意

由于 `Slot` 使用的 `Container` 已被Forge弃用，取而代之的是使用 `IItemHandler` 功能，因此其余解释将围绕使用功能变体： `SlotItemHandler` 展开。

`SlotItemHandler` 包含四个参数： `IItemHandler` 表示物品栈所在的物品栏，该Slot具体表示的物品栈索引，以及该Slot左上角将在屏幕上呈现的相对于 `AbstractContainerScreen#leftPos` 和 `#topPos` 的x和y位置。客户端菜单构造函数应该始终提供相同大小的物品栏的空实例。

在大多数情况下，菜单中包含的任何Slot都会首先添加，然后是玩家的物品栏，最后以玩家的快捷栏结束。要从菜单中访问任何单独的 `Slot` ，必须根据添加Slot的顺序计算索引。

`DataSlot` 是一个抽象类，它应该实现getter和setter来引用存储在数据存储对象中的数据。客户端菜单构造函数应始终通过 `DataSlot#standalone` 提供一个新实例。

每次初始化新菜单时，都应该重新创建上述内容以及Slot。

警告

尽管 `DataSlot` 存储一个整数（int），但由于它在网络上发送数值的方式，它实际上被限制为**short**类型（-32768到32767）。该整数（int）的16个高比特位被忽略。

```
// 假设我们有一个来自大小为5的数据对象的物品栏
// 假设我们在每次初始化服务端菜单时都构造了一个DataSlot

// 客户端菜单构造函数
public MyMenuAccess(int containerId, Inventory playerInventory) {
  this(containerId, playerInventory, new ItemStackHandler(5), DataSlot.standalone());
}

// 服务端菜单构造函数
public MyMenuAccess(int containerId, Inventory playerInventory, IItemHandler dataInventory, DataSlot dataSingle) {
  // 检查数据物品栏大小是否为某个固定值
  // 然后，为数据物品栏添加Slot
  this.addSlot(new SlotItemHandler(dataInventory, /*...*/));

  // 为玩家物品栏添加Slot
  this.addSlot(new Slot(playerInventory, /*...*/));

  // 为被处理的整数添加Slot
  this.addDataSlot(dataSingle);

  // ...
}
```

`ContainerData`

如果需要将多个整数同步到客户端，则可以使用一个 `ContainerData` 来引用这些整数。此接口用作索引查找，以便每个索引表示不同的整数。如果通过 `#addDataSlots` 将 `ContainerData` 添加到菜单中，则也可以在数据对象本身中构造 `ContainerData` 。该方法为接口指定量的数据创建一个新的 `DataSlot` 。客户端菜单构造函数应始终通过 `SimpleContainerData` 提供一个新实例。

```
// 假设我们有一个大小为3的ContainerData

// 客户端菜单构造函数
public MyMenuAccess(int containerId, Inventory playerInventory) {
  this(containerId, playerInventory, new SimpleContainerData(3));
}

// 服务端菜单构造函数
public MyMenuAccess(int containerId, Inventory playerInventory, ContainerData dataMultiple) {
  // 检查ContainerData大小是否为某个固定值
  checkContainerDataCount(dataMultiple, 3);

  // 为被处理的整数添加Slot
  this.addDataSlots(dataMultiple);

  // ...
}
```

> **警告**
>
> 由于 `ContainerData` 委托 `DataSlot` ，这些整数也被限制为**short**（-32768到32767）。

`#quickMoveStack`

`#quickMoveStack` 是任何菜单都必须实现的第二个方法。每当物品栈被Shift单击或快速移出其当前Slot，直到物品栈完全移出其上一个Slot，或者物品栈没有其他位置可去时，就会调用此方法。该方法返回正在快速移动的Slot中物品栈的一个副本。

物品栈通常使用 `#moveItemStackTo` 在Slot之间移动，它将物品栈移动到第一个可用的Slot中。它接受要移动的物品栈、尝试将物品栈移动到的第一个Slot的索引（包括）、最后一个Slot的索引，以及是以从第一个到最后一个（当 `false` 时）还是从最后一个到第一个（当 `true` 时）的顺序检查Slot。

在Minecraft的实现中，这种方法的逻辑相当一致：

```
// 假设我们有一个大小为5的数据物品栏
// 该物品栏有4个输入（索引1 - 4）并输出到一个结果Slot（索引0）
// 我们也有27个玩家物品栏Slot和9个快捷栏Slot
// 这样，真正的Slot索引按如下编排：
//   - 数据物品栏：结果（0），输入（1 - 4）
//   - 玩家物品栏（5 - 31）
//   - 玩家快捷栏（32 - 40）
@Override
public ItemStack quickMoveStack(Player player, int quickMovedSlotIndex) {
  // 快速移动的Slot的物品栈
  ItemStack quickMovedStack = ItemStack.EMPTY;
  // 快速移动的Slot
  Slot quickMovedSlot = this.slots.get(quickMovedSlotIndex)

  // 如果该Slot在合理范围内且不为空
  if (quickMovedSlot != null && quickMovedSlot.hasItem()) {
    // 获取原始物品栈以用于移动
```

```
ItemStack rawStack = quickMovedSlot.getItem();
// 将Slot物品栈设置为该原始物品栈的副本
quickMovedStack = rawStack.copy();

/*
以下快速移动逻辑可以简化为：如果在数据物品栏中，尝试移动到玩家物品栏/快捷栏，
反之亦然，对于无法转换数据的容器（例如箱子）。
*/

// 如果快速移动在数据物品栏的结果Slot上进行
if (quickMovedSlotIndex == 0) {
  // 尝试将结果Slot移入玩家物品栏/快捷栏
  if (!this.moveItemStackTo(rawStack, 5, 41, true)) {
    // 如果无法移动，就不再进行快速移动
    return ItemStack.EMPTY;
  }

  // 执行Slot的快速移动逻辑
  slot.onQuickCraft(rawStack, quickMovedStack);
}
// 否则如果快速移动在玩家物品栏或快捷栏Slot上进行
else if (quickMovedSlotIndex >= 5 && quickMovedSlotIndex < 41) {
  // 尝试将物品栏/快捷栏Slot移入数据物品栏输入Slot
  if (!this.moveItemStackTo(rawStack, 1, 5, false)) {
    // 如果无法移动且在玩家物品栏Slot内，尝试移入快捷栏
    if (quickMovedSlotIndex < 32) {
      if (!this.moveItemStackTo(rawStack, 32, 41, false)) {
        // 如果无法移动，就不再进行快速移动
        return ItemStack.EMPTY;
      }
    }
    // 否则就尝试将快捷栏移入玩家物品栏Slot
    else if (!this.moveItemStackTo(rawStack, 5, 32, false)) {
      // 如果无法移动，就不再进行快速移动
      return ItemStack.EMPTY;
    }
  }
}
// 否则如果快速移动在数据物品栏的输入Slot上进行，尝试将其移入玩家物品栏/快捷栏
else if (!this.moveItemStackTo(rawStack, 5, 41, false)) {
  // 如果无法移动，就不再进行快速移动
  return ItemStack.EMPTY;
}

if (rawStack.isEmpty()) {
  // 如果原始物品栈已完全移出当前Slot，将该Slot置空
  quickMovedSlot.set(ItemStack.EMPTY);
} else {
  // 否则，通知该Slot物品栈数量已改变
  quickMovedSlot.setChanged();
}

/*
如果菜单不表示可以转换物品栈的容器（例如箱子），则可以删除以下if语句和
Slot#onTake调用。
*/
if (rawStack.getCount() == quickMovedStack.getCount()) {
  // 如果原始物品栈不能被移动到另一个Slot，就不再进行快速移动
  return ItemStack.EMPTY;
```

```
    }
    // 执行剩余物品栈的移动后逻辑
    quickMovedSlot.onTake(player, rawStack);
  }

  return quickMovedStack; // 返回该Slot物品栈
}
```

11.1.3 打开菜单

一旦注册了菜单类型，菜单本身已经完成，并且一个屏幕（Screen）已被附加，玩家就可以打开菜单。可以通过在逻辑服务端上调用 `NetworkHooks#openScreen` 来打开菜单。该方法让玩家打开菜单，服务端端菜单的 `MenuProvider` ，如果需要将额外数据同步到客户端，还可以选择 `FriendlyByteBuf` 。

注意

只有在使用 `IContainerFactory` 创建菜单类型时，才应使用带有 `FriendlyByteBuf` 参数的 `NetworkHooks#openScreen` 。

`MenuProvider`

`MenuProvider` 是一个包含两个方法的接口： `#createMenu` 和 `#getDisplayName` ，前者创建菜单的服务端实例，后者返回一个包含要传递到屏幕（Screen）的菜单标题的组件。 `#createMenu` 方法包含三个参数：菜单的容器id、打开菜单的玩家的物品栏以及打开菜单的玩家。

使用 `SimpleMenuProvider` 可以很容易地创建 `MenuProvider` ，它采用方法引用来创建服务端菜单和菜单标题。

```
// 在某种实现中
NetworkHooks.openScreen(serverPlayer, new SimpleMenuProvider(
  (containerId, playerInventory, player) -> new MyMenu(containerId, playerInventory),
  Component.translatable("menu.title.examplemod.mymenu")
));
```

常见的实现

菜单通常在某种玩家交互时打开（例如，当右键单击方块或实体时）。

方块的实现

方块块通常通过重写 `BlockBehaviour#use` 来实现菜单。如果在逻辑客户端上，则交互返回 `InteractionResult#SUCCESS` 。否则，它将打开菜单并返回 `InteractionResult#CONSUME` 。

应通过重写 `BlockBehaviour#getMenuProvider` 来实现 `MenuProvider` 。原版方法使用这个来显示旁观者模式下的菜单。

```
// 在某个Block的子类中
@Override
public MenuProvider getMenuProvider(BlockState state, Level level, BlockPos pos) {
  return new SimpleMenuProvider(/* ... */);
}

@Override
public InteractionResult use(BlockState state, Level level, BlockPos pos, Player player, InteractionHand hand, BlockHitResult result
  if (!level.isClientSide && player instanceof ServerPlayer serverPlayer) {
    NetworkHooks.openScreen(serverPlayer, state.getMenuProvider(level, pos));
  }
  return InteractionResult.sidedSuccess(level.isClientSide);
}
```

注意

这是实现逻辑的最简单的方法，而不是唯一的方法。如果你希望方块仅在特定条件下打开菜单，则需要提前将一些数据同步到客户端，以便在不满足条件的情况下返回 `InteractionResult#PASS` 或 `#FAIL` 。

生物的实现

Mob通常通过重写 `Mob#mobInteract` 来实现菜单。这与方块实现类似，唯一的区别是 `Mob` 本身应该实现 `MenuProvider` 以支持旁观者模式下的显示。

```
public class MyMob extends Mob implements MenuProvider {
  // ...

  @Override
  public InteractionResult mobInteract(Player player, InteractionHand hand) {
    if (!this.level.isClientSide && player instanceof ServerPlayer serverPlayer) {
      NetworkHooks.openScreen(serverPlayer, this);
    }
    return InteractionResult.sidedSuccess(this.level.isClientSide);
  }
}
```

注意

再次说明，这是实现逻辑的最简单的方法，而不是唯一的方法。

11.2 屏幕（Screens）

屏幕通常是Minecraft中所有图形用户界面（GUI）的基础：接收用户输入，在服务端上验证，并将生成的操作同步回客户端。它们可以与菜单（Menus）相结合，为类似物品栏的视图创建通信网络，也可以是独立的，模组开发者可以通过自己的网络实现来处理。

屏幕由许多部分组成，因此很难完全理解Minecraft中的“屏幕”到底是什么。因此，在讨论屏幕本身之前，本文档将介绍屏幕的每个组件及其应用方式。

11.2.1 相对坐标

每当渲染任何东西时，都需要有一些标识符来指定它将出现的位置。通过大量的抽象，Minecraft的大多数渲染调用都在坐标平面中采用x、y和z值。x值从左到右递增，y从上到下递增，z从远到近递增。但是，坐标并不是固定在指定的范围内。它们可以根据屏幕的大小和选项中指定的比例进行更改。因此，在渲染时必须格外小心，以确保坐标值正确缩放到可更改的屏幕大小。

关于如何将坐标相对化的信息将在屏幕部分中呈现。

重要

如果选择使用固定坐标或不正确地缩放屏幕，则渲染的对象可能看起来很奇怪或错位。检查坐标是否正确相对化的一个简单方法是单击视频设置中的“Gui比例”按钮。在确定GUI渲染的比例时，此值用作显示器宽度和高度的除数。

11.2.2 Gui组件

Minecraft渲染的任何GUI都是 `GuiComponent` 的一个实例。 `GuiComponent` 包含用于渲染最常用对象的最基本方法。这些分为三类：彩色矩形、字符串和纹理。还有一种用于呈现组件片段的附加方法（ `#enableScissor` / `#disableScissor` ）。所有方法都接受一个 `PoseStack` ，该 `PoseStack` 应用必要的转换来正确地渲染应该渲染组件的位置。此外，颜色采用ARGB格式。

彩色矩形

彩色矩形是通过位置颜色着色器绘制的。有三种类型的彩色矩形可以绘制。

首先，有一条彩色的水平和垂直一像素宽的线，分别为 `#hLine` 和 `#vLine` 。 `#hLine` 接受两个x坐标，定义左侧和右侧（包括）、顶部y坐标和颜色。 `#vLine` 接受左侧的x坐标、定义顶部和底部（包括）的两个y坐标以及颜色。

其次，还有 `#fill` 方法，它在屏幕上绘制一个矩形。Line方法在内部调用此方法。其接受左x坐标、上y坐标、右x坐标、下y坐标和颜色。

最后，还有 `#fillGradient` 方法，它绘制一个具有垂直梯度的矩形。这包括右x坐标、下y坐标、左x坐标、上y坐标、z坐标以及底部和顶部的颜色。

字符串

字符串是通过其 `Font` 绘制的，通常由它们自己的普通、透视和偏移模式的着色器组成。可以渲染两种对齐的字符串，每种都有一个后阴影：左对齐字符串（ `#drawString` ）和居中对齐字符串（ `#drawCenteredString` ）。这两者都采用了字符串将被渲染的字体、要绘制的字符串、分别表示字符串左侧或中心的x坐标、顶部的y坐标和颜色。

注意

字符串通常应作为 `Component` 传入，因为它们处理各种用例，包括方法的另外两个重载。

纹理

纹理是通过blitting的方式绘制的，因此方法名为 `#blit` ，为此，它复制图像的比特并将其直接绘制到屏幕上。这些是通过位置纹理着色器绘制的。虽然有许多不同的 `#blit` 重载，但我们只讨论两个静态的 `#blit` 。

第一个静态 `#blit` 取六个整数，并假设渲染的纹理位于256 x 256 PNG文件上。它接受左侧x和顶部y屏幕坐标，PNG中的左侧x和底部y坐标，以及要渲染的图像的宽度和高度。

> **注意**
>
> 必须指定PNG文件的大小，以便可以规范化坐标以获得关联的UV值。

第一个 `#blit` 所调用的另一个静态 `#blit` 将参数扩展为九个整数，仅假设图像位于PNG文件上。它获取左侧x和顶部y屏幕坐标、z坐标（称为blit偏移）、PNG中的左侧x和上部y坐标、要渲染的图像的宽度和高度以及PNG文件的宽度和高。

Blit偏移

渲染纹理时的z坐标通常设置为blit偏移。偏移量负责在查看屏幕时对渲染进行适当分层。z坐标较小的渲染在背景中渲染，反之亦然，z坐标较大的渲染在前景中渲染。z偏移量可以通过 `#translate` 直接设置在 `PoseStack` 本身上。

> **重要**
>
> 设置blit偏移时，必须在渲染对象后重置它。否则，屏幕内的其他对象可能会在不正确的层中渲染，从而导致图形问题。建议在平移前推动当前姿势，然后在偏移处完成所有渲染后弹出。

11.2.3 Renderable

`Renderable` 本质上是被渲染的对象。其中包括屏幕、按钮、聊天框、列表等。 `Renderable` 只有一个方法： `#render` 。这需要 `PoseStack` 保存任何先前的变换，以正确渲染可渲染的、缩放到相对屏幕大小的鼠标的x和y位置，以及游戏刻增量（自上一帧以来经过了多少游戏刻）。

一些常见的可渲染文件是屏幕和“小部件”：通常在屏幕上渲染的可交互元素，如 `Button` 、其子类型 `ImageButton` 和用于在屏幕上输入文本的 `EditBox` 。

11.2.4 GuiEventListener

在Minecraft中呈现的任何屏幕都实现了 `GuiEventListener` 。 `GuiEventListener` 负责处理用户与屏幕的交互。其中包括来自鼠标（移动、单击、释放、拖动、滚动、鼠标悬停）和键盘（按下、释放、键入）的输入。每个方法都返回关联的操作是否成功影响了屏幕。按钮、聊天框、列表等小工具也实现了这个界面。

ContainerEventHandler

与 `GuiEventListener` 几乎同义的是它们的子类型： `ContainerEventHandler` 。它们负责处理包含小部件的屏幕上的用户交互，管理当前聚焦的内容以及相关交互的应用方式。 `ContainerEventHandler` 添加了三个附加功能：可交互的子项、拖动和聚焦。

事件处理器包含用于确定元素交互顺序的子级。在鼠标事件处理器（不包括拖动）期间，鼠标悬停的列表中的第一个子级将执行其逻辑。

用鼠标拖动元素，通过 `#mouseClicked` 和 `#mouseReleased` 实现，可以提供更精确的执行逻辑。

聚焦允许在事件执行期间，例如在键盘事件或拖动鼠标期间，首先检查并处理特定的子项。焦点通常通过 `#setFocused` 设置。此外，可以使用 `#nextFocusPath` 循环可交互的子级，根据传入的 `FocusNavigationEvent` 选择子级。

> **注意**
>
> 屏幕通过 `AbstractContainerEventHandler` 实现了 `ContainerEventHandler` 和 `GuiComponent`，添加了setter和getter逻辑用于拖动和聚焦子级。

11.2.5 NarratableEntry

`NarratableEntry` 是可以通过Minecraft的无障碍讲述功能进行讲述的元素。每个元素可以根据悬停或选择的内容提供不同的叙述，通常按焦点、悬停以及所有其他情况进行优先级排序。

`NarratableEntry` 有三种方法：一种是确定元素的优先级（`#narrationPriority`），一种是决定是否说出讲述（`#isActive`），最后一种是将讲述提供给相关的输出（说出或读取）（`#updateNarration`）。

> **注意**
>
> Minecraft中的所有小部件都是 `NarratableEntry`，因此如果使用可用的子类型，通常不需要手动实现。

11.2.6 屏幕子类型

利用以上所有知识，可以构建一个简单的屏幕。为了更容易理解，屏幕的组件将按通常遇到的顺序提及。

首先，所有屏幕都包含一个 `Component`，其表示屏幕的标题。此组件通常由其子类型之一绘制到屏幕上。它仅用于讲述消息的基本屏幕。

```
// 在某个Screen子类中
public MyScreen(Component title) {
    super(title);
}
```

初始化

一旦屏幕被初始化，就会调用 `#init` 方法。 `#init` 方法将屏幕内的初始设置从 `ItemRenderer` 和 `Minecraft` 实例设置为游戏缩放的相对宽度和高度。任何设置，如添加小部件或预计算相对坐标，都应该用这种方法完成。如果调整游戏窗口的大小，屏幕将通过调用 `#init` 方法重新初始化。

有三种方法可以将小部件添加到屏幕中，每种方法都有各自的用途：

方法	描述
`#addWidget`	添加一个可交互和讲述但不被渲染的小部件。
`#addRenderableOnly`	添加一个只会被渲染的小部件；它既不可互动，也不可被讲述。
`#addRenderableWidget`	添加一个可交互、讲述和被渲染的小部件。

通常， `#addRenderableWidget` 将是最常用的。

```
// 在某个Screen子类中
@Override
protected void init() {
    super.init();

    // 添加小部件和已预计算的值
```

```
    this.addRenderableWidget(new EditBox(/* ... */));
}
```

计时屏幕

屏幕也会使用 `#tick` 方法计时来执行某种级别的客户端逻辑以进行渲染。最常见的例子是 `EditBox` 的光标闪烁。

```
// 在某个Screen子类中
@Override
public void tick() {
    super.tick();

    // 在editBox中为EditBox添加计时逻辑
    this.editBox.tick();
}
```

输入处理

由于屏幕是 `GuiEventListener` 的子类型，输入处理器也可以被覆盖，例如用于处理特定按键上的逻辑。

屏幕的渲染

最后，屏幕是通过作为 `Renderable` 子类型提供的 `#render` 方法进行渲染的。如前所述， `#render` 方法绘制屏幕必须渲染每一帧的所有内容，如背景、小部件、提示文本等。默认情况下， `#render` 方法仅将小部件渲染到屏幕上。

在通常不由子类型处理的屏幕中渲染的两件最常见的事情是背景和提示文本。

背景可以使用 `#renderBackground` 进行渲染，其中一种方法在无法渲染屏幕后面的级别时，每当渲染屏幕时，都会将v偏移值作为选项背景。

提示文本通过 `#renderTooltip` 或 `#renderComponentTooltip` 进行渲染，它们可以接受正在渲染的文本组件、可选的自定义提示文本示组件以及提示文本应在屏幕上渲染的x/y相对坐标。

```
// 在某个Screen子类中

// mouseX和mouseY指示鼠标光标在屏幕上的缩放坐标
@Override
public void render(PoseStack pose, int mouseX, int mouseY, float partialTick) {
    // 通常首先渲染背景
    this.renderBackground(pose);

    // 在此处渲染在小部件之前渲染的内容（背景纹理）

    // 然后是窗口小部件，如果这是Screen的直接子项
    super.render(pose, mouseX, mouseY, partialTick);

    // 在小部件之后渲染的内容（工具提示）
}
```

屏幕的关闭

当屏幕关闭时，有两种方法处理屏幕的关闭： `#onClose` 和 `#removed` 。

每当用户做出关闭当前屏幕的输入时，就会调用 `#onClose` 。此方法通常用作回调，以销毁和保存屏幕本身中的任何内部进程。这包括向服务端发送数据包。

`#removed` 在屏幕更改并被释放到垃圾收集器之前被调用。这将处理任何尚未重置回屏幕打开前初始状态的内容。

```
// 在某个Screen子类中

@Override
public void onClose() {
    // 在此处停止任何处理器

    // 最后调用，以防干扰重写后的方法体
    super.onClose();
}

@Override
public void removed() {
    // 在此处重置初始状态

    // 最后调用，以防干扰重写后的方法体
    super.removed()
;}
```

11.2.7 AbstractContainerScreen

如果一个屏幕直接连接到菜单（Menu），那么其应改为继承 `AbstractContainerScreen` 。 `AbstractContainerScreen` 充当菜单的渲染器和输入处理程序，包含用于与Slot同步和交互的逻辑。因此，通常只需要重写或实现两个方法就可以拥有一个可工作的容器屏幕。同样，为了更容易理解，容器屏幕的组件将按通常遇到的顺序提及。

`AbstractContainerScreen` 通常需要三个参数：打开的容器菜单（用泛型 `T` 表示）、玩家物品栏（仅用于显示名称）和屏幕本身的标题。在这里，可以设置多个定位字段：

字段	描述
`imageWidth`	用于背景的纹理的宽度。这通常位于256 x 256的PNG中，默认值为176。
`imageHeight`	用于背景的纹理的高度。这通常位于256 x 256的PNG中，默认值为166。
`titleLabelX`	将渲染屏幕标题的位置的相对x坐标。
`titleLabelY`	将渲染屏幕标题的位置的相对y坐标。
`inventoryLabelX`	将渲染玩家物品栏名称的位置的相对x坐标。
`inventoryLabelY`	将渲染玩家物品栏名称的位置的相对y坐标。

重要

在上一节中提到应该在`#init`方法中设置预先计算的相对坐标。这仍然保持正确，因为这里提到的值不是预先计算的坐标，而是静态值和相对坐标。

图像值是静态的且不变，因为它们表示背景纹理大小。为了在渲染时更容易，在`#init`方法中预先计算了两个附加值（`leftPos`和`topPos`），该方法标记了将渲染背景的左上角。标签坐标相对于这些值。

`leftPos`和`topPos`也被用作渲染背景的方便方式，因为它们已经表示要传递到`#blit`方法中的位置。

```
// 在某个AbstractContainerScreen子类中
public MyContainerScreen(MyMenu menu, Inventory playerInventory, Component title) {
    super(menu, playerInventory, title);

    this.titleLabelX = 10;
    this.inventoryLabelX = 10;

    /*
     * 如果'imageHeight'已更改，则还必须更改'inventoryLabelY'，因为该值取决于'imageHeight'值。
     */
}
```

屏幕的访问

当菜单被传递给屏幕时，菜单中的任何值（通过Slot、数据Slot或自定义系统）都可以通过`menu`字段访问。

容器的计时

当玩家活着并通过`#containerTick`查看屏幕时，容器屏幕在`#tick`方法中计时。这基本上取代了容器屏幕中的`#tick`，其最常见的用法是在配方书中计时。

```
// 在某个AbstractContainerScreen子类中
@Override
protected void containerTick() {
    super.containerTick();

    // 在此处对某些事计时
}
```

容器屏幕的渲染

容器屏幕通过三种方法进行渲染：`#renderBg`，用于渲染背景纹理；`#renderLabels`，用于在背景顶部渲染任何文本；以及`#render`，除了提供灰色背景和提示文本外，还包含前两种方法。

从`#render`开始，最常见的重写（通常是唯一的情况）是添加背景，调用super来渲染容器屏幕，以及最后在其顶部渲染提示文本。

```
// 在某个AbstractContainerScreen子类中
@Override
public void render(PoseStack pose, int mouseX, int mouseY, float partialTick) {
    this.renderBackground(pose);
    super.render(pose, mouseX, mouseY, partialTick);

    /*
     * 该方法由容器屏幕添加，用于渲染悬停在其上的任何Slot的提示文本。
     */
```

```
    this.renderTooltip(pose, mouseX, mouseY);
}
```

在super中， `#renderBg` 被调用以渲染屏幕的背景。最标准的代表是使用三个方法调用：两个用于设置，一个用于绘制背景纹理。

```
// 在某个AbstractContainerScreen子类中

// 背景纹理的位置（assets/<namespace>/<path>）
private static final ResourceLocation BACKGROUND_LOCATION = new ResourceLocation(MOD_ID, "textures/gui/container/my_container_screer

@Override
protected void renderBg(PoseStack pose, float partialTick, int mouseX, int mouseY) {
    /*
     * 设置着色器要使用的纹理位置。虽然最多可以设置12个纹理，但'blit'中
     * 使用的着色器仅查看第一个纹理索引。
     */
    RenderSystem.setShaderTexture(0, BACKGROUND_LOCATION);

    /*
     * 将背景纹理渲染到屏幕上。'leftPos'和'topPos'应该已经表示纹理应该渲染
     * 的左上角，因为它是根据'imageWidth'和'imageHeight'预计算的。两个零
     * 表示256 x 256 PNG文件中的整数u/v坐标。
     */
    GuiComponent.blit(pose, this.leftPos, this.topPos, 0, 0, this.imageWidth, this.imageHeight);
}
```

最后，调用 `#renderLabels` 来渲染背景上方但提示文本下方的任何文本。这个简单的调用使用字体来绘制相关的组件。

```
// 在某个AbstractContainerScreen子类中
@Override
protected void renderLabels(PoseStack pose, int mouseX, int mouseY) {
    super.renderLabels(pose, mouseX, mouseY);

    // 假设我们有个组件'label'
    // 'label'在'labelX'和'labelY'处被绘制
    this.font.draw(pose, label, labelX, labelY, 0x404040);
}
```

> **注意**
>
> 渲染标签时，不需要指定 `leftPos` 和 `topPos` 偏移量。这些已经在 `PoseStack` 中进行了转换，因此该方法中的所有内容都是相对于这些坐标绘制的。

11.2.8 注册一个AbstractContainerScreen

要将 `AbstractContainerScreen` 与菜单一起使用，需要对其进行注册。这可以通过调用模组事件总线上的 `FMLClientSetupEvent` 中的 `MenuScreens#register` 来完成。

```
// 该事件已在模组事件总线上被监听
private void clientSetup(FMLClientSetupEvent event) {
    event.enqueueWork(
        // 假设：RegistryObject<MenuType<MyMenu>> MY_MENU
        // 假设MyContainerScreen<MyMenu>，其接受三个参数
        () -> MenuScreens.register(MY_MENU.get(), MyContainerScreen::new)
```

```
    );
}
```

> **警告**
>
> `MenuScreens#register` 不是线程安全的，因此它需要在并行调度事件提供的 `#enqueueWork` 内部调用。

12. 渲染

12.1 模型扩展

12.1.1 根变换

在模型JSON的顶层添加 `transform` 条目向加载器建议，在方块模型的情况下，应在方块状态文件中的旋转之前对所有几何体应用变换，在物品模型的情况中，应在显示变换之前对其应用变换。转换可通过 `IUnbakedGeometry#bake()` 中的 `IGeometryBakingContext#getRootTransform()` 获得。

自定义模型加载器可能会完全忽略此字段。

根变换可以用两种格式指定：

1. 一个JSON对象，包含一个奇异的 `matrix` 条目，该条目包含一个嵌套JSON数组形式的原始转换矩阵，省略了最后一行（3*4矩阵，行主序）。矩阵是按平移、左旋转、缩放、右旋转和变换原点的顺序组成的。结构示例：

```
"transform": {
    "matrix": [
        [ 0, 0, 0, 0 ],
        [ 0, 0, 0, 0 ],
        [ 0, 0, 0, 0 ]
    ]
}
```

2. 一个JSON对象，包含以下可选项的任意组合：
   - `origin`：用于旋转和缩放的原点
   - `translation`：相对平移
   - `rotation` 或 `left_rotation`：在缩放之前围绕要应用的平移原点旋转
   - `scale`：相对于平移原点的比例
   - `right_rotation` 或 `post_rotation`：在缩放之后要应用的围绕平移原点的旋转

元素的指定

如果转换被指定为选项4中提到的条目的组合，则这些条目将按照 `translation` 、 `left_rotation` 、 `scale` 、 `right_rotation` 的顺序应用。 转换将移动到指定的原点，作为最后一步。

```
{
    "transform": {
        "origin": "center",
        "translation": [ 0, 0.5, 0 ],
        "rotation": { "y": 45 }
    },
    // ...
}
```

这些元素的定义应为如下：

原点

原点可以指定为表示三维矢量的3个浮点数的数组：`[ x, y, z ]`，也可以指定为三个默认值之一：

- `"corner"` (0, 0, 0)
- `"center"` (.5, .5, .5)
- `"opposing-corner"` (1, 1, 1)

如果未指定原点，则其默认为 `"opposing-corner"` 。

平移

平移必须指定为表示三维矢量的3个浮点数的数组：`[ x, y, z ]`，如果不存在，则默认为(0, 0, 0)。

左旋转和右旋转

可以通过以下四种方式中的任何一种指定旋转：

- 具有单轴=>旋转度映射的单个JSON对象：`{ "x": 90 }`
- 具有上述格式的任意数量的JSON对象的数组（按指定顺序应用）：`[ { "x": 90 }, { "y": 45 }, { "x": -22.5 } ]`
- 由3个浮点数组成的数组，指定围绕每个轴的旋转（以度为单位）：`[ 90, 180, 45 ]`
- 直接指定四元数的4个浮点数的数组：`[ 0.38268346, 0, 0, 0.9238795 ]`（示例等于绕X轴45度）

如果未指定相应的旋转，则默认为无旋转。

比例

比例必须指定为表示三维矢量的3个浮点数的数组：`[ x, y, z ]`，如果不存在，则默认为(1, 1, 1)。

12.1.2 渲染类型

在JSON的顶层添加 `render_type` 条目会向加载器建议模型应该使用什么渲染类型。如果未指定，加载器将选择所使用的渲染类型，通常会回到 `ItemBlockRenderTypes#getRenderLayers()` 返回的渲染类型。

自定义模型加载器可能会完全忽略此字段。

注意

自1.19以来，这比不推荐的通过 `ItemBlockRenderTypes#setRenderLayer()` 为方块设置适用渲染类型的方法更可取。

具有玻璃纹理的透明方块的模型示例

```
{
  "render_type": "minecraft:cutout",
  "parent": "block/cube_all",
  "textures": {
    "all": "block/glass"
  }
}
```

原版值

Forge提供了以下具有相应区块和实体渲染类型的选项（`NamedRenderTypeManager#preRegisterVanillaRenderTypes()`）：

- `minecraft:solid`
  - 区块渲染类型：`RenderType#solid()`
  - 实体渲染类型：`ForgeRenderTypes#ITEM_LAYERED_SOLID`
  - 用于完全固体方块（即石头）
- `minecraft:cutout`
  - 区块渲染类型：`RenderType#cutout()`
  - 实体渲染类型：`ForgeRenderTypes#ITEM_LAYERED_CUTOUT`
  - 用于任何给定像素完全透明或完全不透明的方块（即玻璃方块）
- `minecraft:cutout_mipped`
  - 区块渲染类型：`RenderType#cutoutMipped()`
  - 实体渲染类型：`ForgeRenderTypes#ITEM_LAYERED_CUTOUT`
  - 方块和实体渲染类型不同，因为实体渲染类型上的mipmapping使物品看起来很奇怪
  - 用于任何给定像素是完全透明或完全不透明的方块，并且纹理应在较大距离上缩小（mipmapping）以避免视觉伪影（即树叶）
- `minecraft:cutout_mipped_all`
  - 区块渲染类型：`RenderType#cutoutMipped()`
  - 实体渲染类型：`ForgeRenderTypes#ITEM_LAYERED_CUTOUT_MIPPED`
  - 在类似于 `minecraft:cutout_mipped` 的情况下使用，此时物品表示也应应用mipmapping
- `minecraft:translucent`
  - 区块渲染类型：`RenderType#translucent()`
  - 实体渲染类型：`ForgeRenderTypes#ITEM_LAYERED_TRANSLUCENT`
  - 用于任何给定像素可能部分透明的方块（即彩色玻璃）
- `minecraft:tripwire`
  - 区块渲染类型：`RenderType#tripwire()`
  - 实体渲染类型：`ForgeRenderTypes#ITEM_LAYERED_TRANSLUCENT`
  - 区块和实体渲染类型不同，因为绊线渲染类型作为实体渲染类型不可行
  - 用于具有渲染到天气渲染目标（即绊线）的特殊要求的块

自定义值

要在模型中指定的自定义命名渲染类型可以在 `RegisterNamedRenderTypesEvent` 中注册。此事件在模组事件总线上激发。

自定义命名渲染类型由两个或三个组件组成：

- 区块渲染类型-可以使用 `RenderType.chunkBufferLayers()` 返回的列表中的任何类型
- 具有 `DefaultVertexFormat.NEW_ENTITY` 顶点格式的渲染类型（“实体渲染类型”）
- 具有 `DefaultVertexFormat.NEW_ENTITY` 顶点格式的渲染类型，用于当*Fabulous!*图形模式被选择时（可选）

当使用此命名渲染类型的区块被渲染为块几何体的一部分时，将使用区块渲染类型。 当使用此命名渲染类型的物品在Fast和Fancy图形模式（物品栏、地面、物品框架等）中渲染时，将使用需求实体渲染类型。 选择 *Fabulous!*图形模式时，可选实体渲染类型的使用方式与需求实体渲染类型相同。如果需求实体渲染类型在*Fabulous!*图形模式下不起作用（通常仅适用于半透明渲染类型），则需要使用此渲染类型。

```
public static void onRegisterNamedRenderTypes(RegisterNamedRenderTypesEvent event)
{
  event.register("special_cutout", RenderType.cutout(), Sheets.cutoutBlockSheet());
```

```
  event.register("special_translucent", RenderType.translucent(), Sheets.translucentCullBlockSheet(), Sheets.translucentItemSh
}
```

然后，这些可以在JSON中寻址为 `<your_mod_id>:special_cutout` 和 `<your_mod_id>:special_translucent` 。

12.1.3 部分可见度

在模型JSON的顶层添加 `visibility` 条目可以控制模型不同部分的可见性，以决定是否应将它们烘焙到最终的 `BakedModel` 中。“零件”的定义取决于加载此模型的模型加载器，自定义模型加载器可以完全忽略此条目。在Forge提供的模型加载器中，只有复合模型加载器和OBJ模型加载器使用了此功能。可见性条目被指定为 `"part name": boolean` 条目。

具有两个部分的复合模型的示例，其中第二个部分不会烘焙到最终模型中，并且两个子模型覆盖此可见性，分别只显示第一个部分和两个部分：

```
// mycompositemodel.json
{
  "loader": "forge:composite",
  "children": {
    "part_one": {
      "parent": "mymod:mypartmodel_one"
    },
    "part_two": {
      "parent": "mymod:mypartmodel_two"
    }
  },
  "visibility": {
    "part_two": false
  }
}

// mycompositechild_one.json
{
  "parent": "mymod:mycompositemodel",
  "visibility": {
    "part_one": false,
    "part_two": true
  }
}

// mycompositechild_two.json
{
  "parent": "mymod:mycompositemodel",
  "visibility": {
    "part_two": true
  }
}
```

给定部分的可见性是通过检查模型是否指定了该部分的可见性来确定的，如果不存在，则递归地检查模型的父级，直到找到条目或没有其他父级要检查，在这种情况下，它默认为true。

这允许进行以下设置，其中多个模型使用单个复合模型的不同部分：

1. 复合模型指定多个组件
2. 多个模型将此复合模型指定为其父模型
3. 这些子模型分别指定部分的不同可见性

12.1.4 面数据

在原版的“元素”模型中，可以在元素级别或面级别指定有关元素面的附加数据。未指定自己的面数据的面将返回到元素的面数据，或者如果在元素级别未指定面数据，则返回到默认值。

要将此扩展用于生成的物品模型，必须通过 `forge:item_layers` 模型加载程序加载该模型，因为原版物品模型生成器没有扩展为读取此附加数据。

面数据的全部值都是可选的。

元素模型

在原版的“元素”模型中，面数据应用于指定它的面，或者指定它的元素中没有自己的面数据的所有面。

注意

如果在面上指定了 `forge_data`，它将不会继承元素级 `forge_data` 声明中的任何参数。

可以通过本例中展示的两种方式指定附加数据：

```
{
  "elements": [
    {
      "forge_data": {
        "color": "0xFFFF0000",
        "block_light": 15,
        "sky_light": 15,
        "ambient_occlusion": false
      },
      "faces": {
        "north": {
          "forge_data": {
            "color": "0xFFFF0000",
            "block_light": 15,
            "sky_light": 15,
            "ambient_occlusion": false
          },
          // ...
        },
        // ...
      },
      // ...
    }
  ]
}
```

生成的物品模型

在使用 `forge:item_layers` 加载程序生成的物品模型中，为每个纹理层指定面数据，并应用于所有几何体（前/后向四边形和边四边形）。

`forge_data` 字段必须位于模型JSON的顶层，每个键值对将人脸数据对象与层索引相关联。

在以下示例中，层1将着色为红色并以全亮度发光：

```
{
  "textures": {
```

```
    "layer0": "minecraft:item/stick",
    "layer1": "minecraft:item/glowstone_dust"
  },
  "forge_data": {
    "1": {
      "color": "0xFFFF0000",
      "block_light": 15,
      "sky_light": 15,
      "ambient_occlusion": false
    }
  }
}
```

参数

颜色

使用 `color` 条目指定颜色值将该颜色作为色调应用于四边形。默认值为 `0xFFFFFFFF` （白色，完全不透明）。颜色必须是压缩为32位整数的 `ARGB` 格式，并且可以指定为十六进制字符串（ `"0xAARRGGBB"` ）或十进制整数文字（JSON不支持十六进制整数文字）。

> **警告**
>
> 四种颜色分量与纹理的像素相乘。省略alpha分量相当于将其设为0，这将使几何体完全透明。

如果颜色值为常量，则可以用 `BlockColor` 和 `ItemColor` 替换着色。

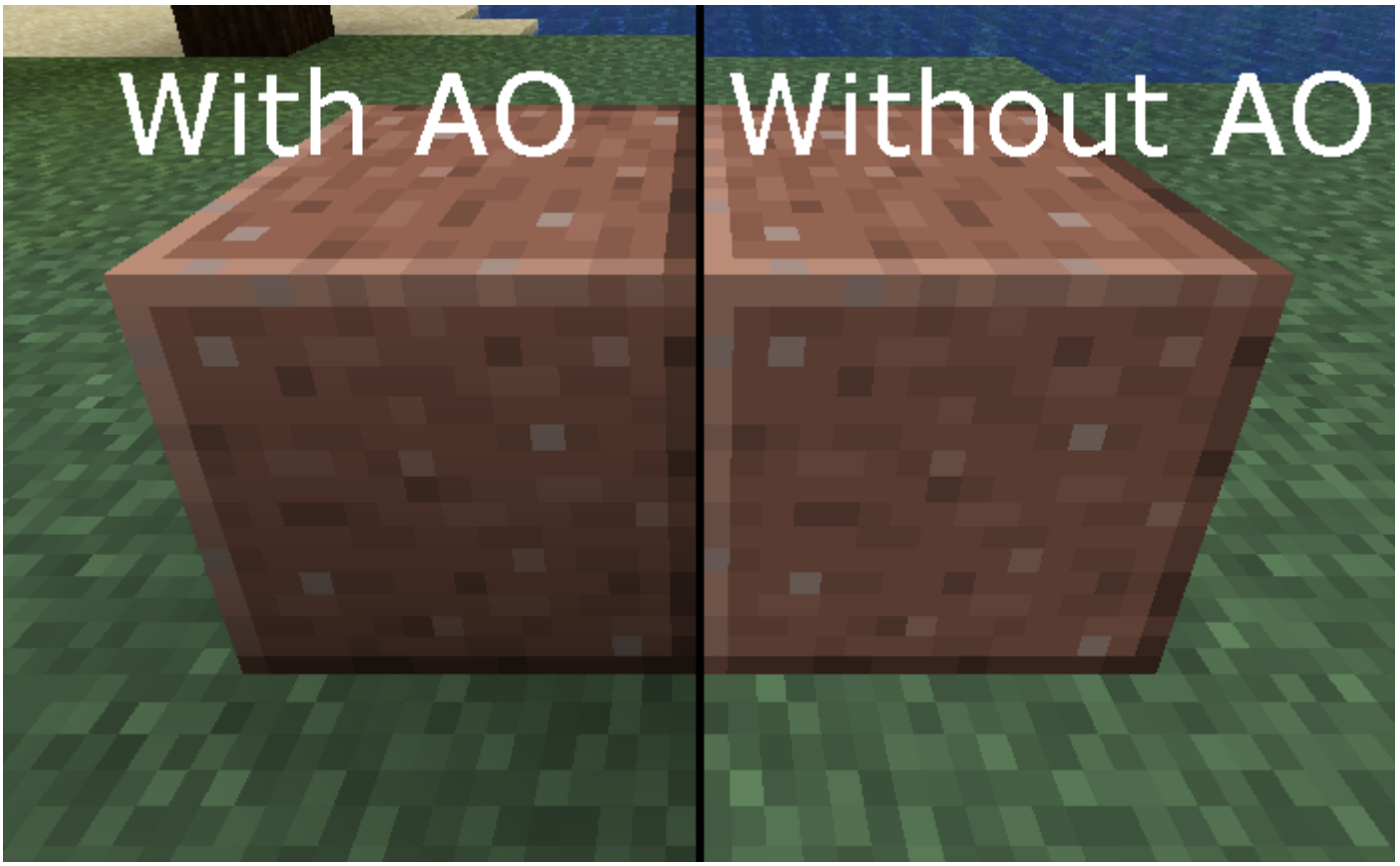
方块亮度和天空亮度

分别使用 `block_light` 和 `sky_light` 条目指定方块和/或天空的亮度值将覆盖四边形的相应亮度值。两个值都默认为0。这些值必须在0-15（包括0-15）的范围内，并且在渲染面时被视为相应光照类型的最小值，这意味着相应光照类型在世界中的较高值将覆盖指定值。

指定的亮度值纯粹是客户端的，既不影响服务器的亮度级别，也不影响周围方块的亮度。

环境光遮挡

指定 `ambient_occlusion` 标志将为四边形配置环境光遮挡（AO）。默认为 `true` 。该标志的行为相当于原版格式的顶级 `ambientocclusion` 标志。



环境光遮挡在左侧启用，在右侧禁用，通过“平滑光照”图形设置演示

注意

如果顶级AO标志设置为false，则在元素或面上将该标志指定为true将无法覆盖顶级标志。

```
{
  "ambientocclusion": false,
  "elements": [
    {
      "forge_data": {
        "ambient_occlusion": true // 无效
      }
    }
  ]
}
```

12.2 模型加载器

12.2.1 自定义模型加载器

“模型”只是一种形状。它可以是一个简单的立方体，可以是几个立方体，也可以是截角二十面体，或者介于两者之间的任何东西。你将看到的大多数模型都是普通的JSON格式。其他格式的模型在运行时由 `IGeometryLoader` 加载到 `IUnbakedGeometry` 中。Forge为WaveFront OBJ文件、bucket、复合模型、不同渲染层中的模型提供了默认实现，并重新实现了原版的 `builtin/generated` 物品模型。大多数事情都不关心加载了什么模型或模型的格式，因为它们最终都由代码中的 `BakedModel` 表示。

> **警告**
>
> 通过模型JSON中的顶级 `loader` 条目指定自定义模型加载程序将导致 `elements` 条目被忽略，除非它被自定义加载程序使用。所有其他普通条目仍将被加载并在未烘焙的 `BlockModel` 表示中可用，并且可能在自定义加载程序之外被使用。

WaveFront OBJ模型

Forge为 `.obj` 文件格式添加了一个加载程序。要使用这些模型，JSON必须引用 `forge:obj` 加载程序。此加载程序接受位于已注册命名空间中且路径以 `.obj` 结尾的任何模型位置。 `.mtl` 文件应放置在与要自动使用的 `.obj` 具有相同名称的相同位置。 `.mtl` 文件可能需要手动编辑才能更改指向JSON中定义的纹理的路径。此外，纹理的V轴可以根据创建模型的外部程序翻转（即，V=0可能是底部边缘，而不是顶部边缘）。这可以在建模程序本身中纠正，也可以在模型JSON中这样做：

```
{
  // 在与'model'声明相同的级别上添加以下行
  "loader": "forge:obj",
  "flip_v": true,
  "model": "examplemod:models/block/model.obj",
  "textures": {
    // 可在.mtl中用#texture0引用
    "texture0": "minecraft:block/dirt",
    "particle": "minecraft:block/dirt"
  }
}
```

12.2.2 烘焙模型（ BakedModel ）

`BakedModel` 是对普通模型加载器调用 `UnbakedModel#bake` 或对自定义模型加载器调用 `IUnbakedGeometry#bake` 的结果。与 `UnbakedModel` 或 `IUnbakedGeometry` 不同， `BakedModel` 纯粹代表一种没有任何物品或方块概念的形状，而不是抽象的。它表示已经优化并简化为可以（几乎）进入GPU的几何体。它还可以处理物品或方块的状态以更改模型。

在大多数情况下，实际上没有必要手动实现此接口。相反，可以使用现有的实现之一。

`getOverrides`

返回要用于此模型的 `ItemOverrides` 。仅当此模型被渲染为物品时才使用此选项。

`useAmbientOcclusion`

如果模型在存档中渲染为方块，则有问题的方块不会发出任何光，并且环境光遮挡处于启用状态。这将导致使用环境光遮挡来渲染模型。

`isGui3d`

如果模型被渲染为物品栏中的物品，在地面上被渲染为实体，在物品框架上，等等，这会使模型看起来“扁平”。在GUI中，这也会禁用照明。

`isCustomRenderer`

> **重要**
>
> 除非你知道自己在做什么，否则只需 `return false` 然后继续其他事项。

将其渲染为物品时，返回 `true` 将导致模型不被渲染，转而回到 `BlockEntityWithoutLevelRenderer#renderByItem` 。对于某些原版物品，如箱子和旗帜，此方法被硬编码为将数据从物品复制到 `BlockEntity` 中，然后使用 `BlockEntityRenderer` 来渲染BE以代替物品。对于所有其他物品，它将使用由 `IClientItemExtensions#getCustomRenderer` 提供的 `BlockEntityWithoutLevelRenderer` 实例。有关详细信息，请参阅BlockEntityWithoutLevelRenderer页。

`getParticleIcon`

粒子应使用的任何纹理。对于方块，它将在实体掉落在其上或其被破坏时显示。对于物品，它将在报废或被吃掉时显示。

> **重要**
>
> 由于模型数据可能会对特定模型的渲染方式产生影响，因此不推荐使用不带参数的原版方法，而推荐使用 `#getParticleIcon(ModelData)` 。

~~`getTransforms`~~

此方法被废弃，推荐实现 `#applyTransform` 。如果实现了 `#applyTransform` ，则该默认实现是足够的。参见变换。

`applyTransform`

参见变换。

`getQuads`

这是 `BakedModel` 的主要方法。它返回一个 `BakedQuad` 的列表：包含将用于渲染模型的低级顶点数据的对象。如果模型被呈现为方块，那么传入的 `BlockState` 是非空的。如果模型被呈现为物品，则从 `#getOverrides` 返回的 `ItemOverrides` 负责处理物品的状态，并且 `BlockState` 参数将为 `null` 。

传入的 `Direction` 用于面剔除。如果正在渲染的另一个方块的给定边上的块是不透明的，则不会渲染与该边关联的面。如果该参数为 `null`，则返回与边不关联的所有面（其永远不会被剔除）。

`rand` 参数是Random的一个实例。

它还接受一个非null的 `ModelData` 实例。这可用于在通过 `ModelProperty` 渲染特定模型时定义额外数据。例如，一个这样的属性是 `CompositeModel$Data`，用于使用 `forge:composite` 模型加载器存储模型的任何附加子模型数据。

请注意，此方法经常被调用：对于一个存档中的每个方块，非剔除面和支持的方块渲染层的每个组合（任何位置从0到28次）调用一次。这给方法应该尽可能快，并且可能需要大量缓存。

12.2.3 变换

当 `BakedModel` 被渲染为物品时，它可以根据在哪个变换中渲染它来应用特殊处理。“变换”是指在什么上下文中渲染模型。可能的转换在代码中由 `ItemDisplayContext` 枚举表示。有两种处理转换的系统：不推荐使用的原版系统，由 `BakedModel#getTransforms` 、 `ItemTransforms` 和 `ItemTransform` 构成；Forge系统，由方法 `IForgeBakedModel#applyTransform` 实现。原版代码被进行了修补，以便尽可能使用 `applyTransform` 而不是原版系统。

ItemDisplayContext

`NONE` - 默认情况下，当未设置上下文时，用于显示实体；当 `Block` 的 `RenderShape` 设置为 `#ENTITYBLOCK_ANIMATED` 时，被Forge使用。

`THIRD_PERSON_LEFT_HAND` / `THIRD_PERSON_RIGHT_HAND` / `FIRST_PERSON_LEFT_HAND` / `FIRST_PERSON_RIGHT_HAND` - 第一人称值表示玩家何时将物品握在自己手中。第三人称值表示当另一个玩家拿着物品，而客户端用第三人称看着它们时。手的含义是不言自明的。

`HEAD` - 表示当任何玩家在头盔槽中佩戴该物品时（例如南瓜）。

`GUI` - 表示当该物品被在一个 `Screen` 中渲染时。

`GROUND` - 表示该物品在存档中作为一个 `ItemEntity` 被渲染时。

`FIXED` - 用于物品展示框。

原版的方式

原版处理转换的方式是通过 `BakedModel#getTransforms` 。此方法返回一个 `ItemTransforms` ，这是一个简单的对象，包含各种作为 `public final` 的 `ItemTransform` 字段。 `ItemTransform` 表示要应用于模型的旋转、平移和比例。 `ItemTransforms` 是这些的容器，除了 `NONE` 之外，每个 `ItemDisplayContext` 都有一个容器。在原版实现中，为 `NONE` 调用 `#getTransform` 会产生默认转换 `ItemTransform#NO_TRANSFORM` 。

Forge废弃了使用处理转换的整个原版系统， `BakedModel` 的大多数实现应该简单地从 `BakedModel#getTransforms` 中 `return ItemTransforms#NO_TRANSFORMS` （这是默认实现）。相反，他们应该实现 `#applyTransform` 。

Forge的方式

Forge处理转换的方法是 `#applyTransform` ，这是一种修补到 `BakedModel` 中的方法。它取代了 `#getTransforms` 方法。

BakedModel#applyTransform

给定一个 `ItemDisplayContext` 、 `PoseStack` 和一个布尔值来确定是否对左手应用变换，此方法将生成一个要渲染的 `BakedModel` 。因为返回的 `BakedModel` 可以是一个全新的模型，所以这种方法比原版方法（例如，一张手里看起来很平但在地上皱巴巴的纸）更灵活。

12.2.4 物品重载（ ItemOverrides ）

`ItemOverrides` 为 `BakedModel` 提供了一种处理 `ItemStack` 状态并返回新 `BakedModel` 的方法；此后，返回的模型将替换旧模型。 `ItemOverrides` 表示任意函数 `(BakedModel, ItemStack, ClientLevel, LivingEntity, int)` → `BakedModel` ，使其适用于动态模型。在原版中，它用于实现物品属性重写。

`ItemOverrides()`

给定 `ItemOverride` 的列表，该构造函数将复制并烘焙该列表。可以使用 `#getOverrides` 访问烘焙后的覆盖。

`resolve`

这需要一个 `BakedModel` 、 `ItemStack` 、 `ClientLevel` 、 `LivingEntity` 和 `int` 来生成另一个用于渲染的 `BakedModel` 。这是模型可以处理其物品状态的地方。

这不应该改变存档。

`getOverrides`

返回一个不可变列表，该列表包含此 `ItemOverrides` 使用的所有 `BakedOverride` 。如果不适用，则返回空列表。

BakedOverride

这个类表示一个原版的物品覆盖，它为一个物品和一个模型的属性保存了几个 `ItemOverrides$PropertyMatcher` ，以备满足这些匹配器时使用。它们是原版物品JSON模型的 `overrides` 数组中的对象：

```
{
  // 在一个原版JSON物品模型内
  "overrides": [
    {
      // 这是一个ItemOverride
      "predicate": {
        // 这是Map<ResourceLocation, Float>，包含属性的名称以及它们的最小值
        "example1:prop": 0.5
      },
      // 这是该覆盖的'location'或目标模型，如果上面的predicate匹配，则使用它
      "model": "example1:item/model"
    },
    {
      // 这是另一个ItemOverride
      "predicate": {
        "example2:prop": 1
      },
      "model": "example2:item/model"
    }
  ]
}
```

13. 资源

13.1 客户端资源（Assets）

13.1.1 资源包

资源包允许通过 `assets` 目录自定义客户端资源。这包括纹理、模型、声音、本地化和其他。你的模组（以及Forge本身）也可以有资源包。因此，任何用户都可以修改该目录中定义的所有纹理、模型和其他资源。

创建一个资源包

资源包存储在项目的资源中。 `assets` 目录包含该包的内容，而该包本身则由 `assets` 文件夹旁边的 `pack.mcmeta` 定义。 你的模组可以有多个资源域，因为你可以添加或修改现有的资源包，比如原版的、Forge的或其他模组的。 然后，你可以按照在Minecraft Wiki中找到的步骤创建任何资源包。

附加阅读：资源位置

13.1.2 模型

模型

模型系统是Minecraft赋予方块和物品形状的方式。通过模型系统，方块和物品被映射到它们的模型，这些模型定义了它们的外观。模型系统的主要目标之一是不仅允许纹理，还允许资源包更改方块/物品的整个形状。事实上，任何添加物品或方块的模组也包含用于其方块和物品的迷你资源包。

模型文件

模型和纹理通过 `ResourceLocation` 链接，但使用 `ModelResourceLocation` 存储在 `ModelManager` 中。模型通过方块或物品的注册表名称在不同位置引用，具体取决于它们是引用方块状态还是物品模型。方块将使其 `ModelResourceLocation` 代表其注册表名称及其当前 `BlockState` 的字符串化版本，而物品将使用其注册表名称后跟 `inventory` 。

> **注意**
>
> JSON模型只支持长方体元素；没有办法表达三角楔或类似的东西。要有更复杂的模型，必须使用另一种格式。

纹理

纹理和模型一样，包含在资源包中，并被称为 `ResourceLocation` 。在《我的世界》中，UV坐标 (0,0)表示左上角。UV总是从0到16。如果纹理较大或较小，则会缩放坐标以进行拟合。纹理也应该是正方形的，纹理的边长应该是2的幂，否则会破坏mipmapping（例如1x1、2x2、8x8、16x16和128x128是好的。不建议使用5x5和30x30，因为它们不是2的幂。5x10和4x8会完全断裂，因为它们不是正方形的。）。只有当纹理是动画化的时，纹理才应该不是正方形。

纹理色调

原版中的许多方块和物品会根据它们的位置或特性（如草）改变其纹理颜色。模型支持在面上指定“色调索引”，这是可以由 `BlockColor` 和 `ItemColor` 处理的整数。有关如何在原版模型中定义色调索引的信息，请参阅wiki。

`BlockColor / ItemColor`

这两个都是单方法接口。 `BlockColor` 接受一个 `BlockState` 、一个（可为空的） `BlockAndTintGetter` 和一个（可为空的） `BlockPos` 。 `ItemColor` 接受一个 `ItemStack` 。它们都采用一个 `int` 参数 `tintIndex` ，它是正在着色的面的色调索引。 它们都返回一个 `int` ，一个颜色乘数。这个 `int` 被视为4个无符号字节，即alpha、red、green和blue，按照从最高有效字节到最低有效字节的顺序。对于着色面上的每个像素，每个颜色通道的值是 `(int)((float) base * multiplier / 255.0)` ，其中 `base` 是通道的原始值， `multiplier` 是颜色乘数的关联字节。 请注意，方块不使用Alpha通道。例如，未着色的草纹理看起来是白色和灰色的。草的 `BlockColor` 和 `ItemColor` 返回颜色乘数，red和blue分量较低，但alpha和green分量较高（至少在温暖的生物群系中），因此当执行乘法时，绿色会被带出，红色/蓝色会减少。

如果物品继承自 `builtin/generated` 模型，则每个层（“layer0”、“layer1”等）都有与其层索引相对应的色调索引。

创建颜色处理器

`BlockColor` 需要注册到游戏的 `BlockColors` 实例中。 `BlockColors` 可以通过 `RegisterColorHandlersEvent$Block` 获取， `BlockColor` 可以通过 `#register` 注册。请注意，这不会导致给定方块的 `BlockItem` 被着色。 `BlockItem` 是物品，需要使用 `ItemColor` 进行着色。

```
@SubscribeEvent
public void registerBlockColors(RegisterColorHandlersEvent.Block event){
  event.register(myBlockColor, coloredBlock1, coloredBlock2, ...);
}
```

`ItemColor` 需要注册到游戏的 `ItemColors` 实例中。 `ItemColors` 可以通过 `RegisterColorHandlersEvent$Item` 获取， `ItemColor` 可以通过 `#register` 注册。此方法也被重载为接受 `Block` ， 它只是将物品 `Block#asItem` 的颜色处理器注册为物品（即方块的 `BlockItem` ）。

```
@SubscribeEvent
public void registerItemColors(RegisterColorHandlersEvent.Item event){
  event.register(myItemColor, coloredItem1, coloredItem2, ...);
}
```

物品属性

物品属性是将物品的“属性”公开给模型系统的一种方式。一个例子是弓，其中最重要的特性是弓被拉了多远。然后，这些信息用于选择弓的模型，创建拉动弓的动画。

物品属性为其注册的每个 `ItemStack` 分配一个特定的 `float` 值，原版物品模型定义可以使用这些值来定义“覆盖”，其中物品默认为某个模型，但如果覆盖匹配，则覆盖该模型并使用另一个模型。它们之所以有用，主要是因为它们是连续的。例如，弓使用物品属性来定义其拉动动画。物品模型由'float'数字谓词决定，它不受限制，但通常在 `0.0F` 和 `1.0F` 之间。这允许资源包为拉弓动画添加他们想要的任意多个模型，而不是在动画中为他们的模型设置四个“槽”。指南针和时钟也是如此。

向物品添加属性

`ItemProperties#register` 用于向某个物品添加属性。 `Item` 参数是要附加属性的物品（例如 `ExampleItems#APPLE` ）。 `ResourceLocation` 参数是所要赋予属性的名称（例如 `new ResourceLocation("pull")` ）。 `ItemPropertyFunction` 是一个函数接口，它接受 `ItemStack` 、它所在的 `ClientLevel` （可以为null）、持有它的 `LivingEntity` （可以是null）和包含持有实体的id的 `int` （可能是 `0` ），返回属性的 `float` 值。对于修改后的物品属性，建议将模组的mod id用作命名空间（例如 `examplemod:property` ，而不仅仅是 `property` ，因为这实际上意味着 `minecraft:property` ）。这些操作应在 `FMLClientSetupEvent` 中完成。 还有另一个方法 `ItemProperties#registerGeneric` 用于向所有物品添加属性，并且它不将 `Item` 作为其参数，因为所有物品都将应用此属性。

重要

使用 `FMLClientSetupEvent#enqueueWork` 执行这些任务，因为 `ItemProperties` 中的数据结构不是线程安全的。

注意

Mojang反对使用 `ItemPropertyFunction` 而推荐使用 `ClampedItemPropertyFunction` 子接口，该子接口将结果夹在 `0` 和 `1` 之间。

覆盖的使用

覆盖的格式可以在wiki上看到，一个很好的例子可以在 `model/item/bow.json` 中找到。为了参考，这里是一个具有 `examplemod:power` 属性的物品的假设例子。如果值不匹配，则默认为当前模型，但如果有多个匹配，则会选择列表中的最后一个匹配。

重要

predicate适用于大于或等于给定值的所有值。

```
{
  "parent": "item/generated",
  "textures": {
    // Default
    "layer0": "examplemod:items/example_partial"
  },
  "overrides": [
    {
      // power >= .75
      "predicate": {
        "examplemod:power": 0.75
      },
      "model": "examplemod:item/example_powered"
    }
```

```
  ]
}
```

下面是支持代码中的一个假设片段。与旧版本（低于1.16.x）不同，这只需要在客户端完成，因为服务端上不存在 `ItemProperties` 。

```
private void setup(final FMLClientSetupEvent event)
{
  event.enqueueWork(() ->
  {
    ItemProperties.register(ExampleItems.APPLE,
      new ResourceLocation(ExampleMod.MODID, "pulling"), (stack, level, living, id) -> {
        return living != null && living.isUsingItem() && living.getUseItem() == stack ? 1.0F : 0.0F;
      });
  });
}
```

13.2 服务端数据（Data）

13.2.1 数据包

在1.13中，Mojang在游戏基底中添加了数据包。它们允许通过 `data` 目录修改逻辑服务端的文件。这包括进度、战利品表（loot_tables）、结构、配方、标签等。Forge和你的模组也可以有数据包。因此，任何用户都可以修改该目录中定义的所有配方、战利品表和其他数据。

创建一个数据包

数据包存储在项目资源的 `data` 目录中。 你的模组可以有多个数据域，因为你可以添加或修改现有的数据包，比如原版的、Forge的或其他模组的。 然后，你可以按照此处的步骤创建任何数据包。

附加阅读：资源位置

13.2.2 配方

配方

配方是一种将一定数量的对象转换为Minecraft世界中其他对象的方法。尽管原版系统纯粹处理物品转换，但整个系统可以扩展为使用程序员创建的任何对象。

由数据驱动的配方

原版中的大多数配方实现都是通过JSON进行数据驱动的。这意味着创建新配方不需要模组，只需要数据包。关于如何创建这些配方并将其放入模组的 `resources` 文件夹的完整列表可以在Minecraft Wiki上找到。

可以在配方书中获得配方，作为完成进度的奖励。配方进度总是以 `minecraft:recipes/root` 为其父项，以免出现在进度屏幕上。获得配方进度的默认标准是检查用户是否已通过一次使用解锁配方或通过某个如 `/recipe` 的命令接收配方：

```
// 在某个配方进度json中
"has_the_recipe": { // 条件标签
  // 如果examplemod:example_recipe被使用，则成功
  "trigger": "minecraft:recipe_unlocked",
  "conditions": {
    "recipe": "examplemod:example_recipe"
  }
}
//...
"requirements": [
  [
    "has_the_recipe"
    // ... 解锁配方所需的其他用逻辑或相连的条件标签
  ]
]
```

由数据驱动的配方及其解锁的进度可以通过 `RecipeProvider` 生成。

配方管理器

配方是通过 `RecipeManager` 加载和存储的。任何与获取可用配方相关的操作都由该管理器负责。有两种重要的方法需要了解：

方法	描述
`getRecipeFor`	获取与当前输入匹配的第一个配方。
`getRecipesFor`	获取与当前输入匹配的所有配方。

每个方法都接受一个 `RecipeType`，表示使用配方的方法（合成、烧炼等），一个保存输入配置的 `Container`，以及与容器一起传递给 `Recipe#matches` 的当前存档。

重要

Forge提供了`RecipeWrapper`实用类，该类扩展了`Container`，用于包装`IItemHandler`，并将其传递给需要`Container`参数的方法。

```
// 在具有IItemHandlerModifiable处理器的某给方法中
recipeManger.getRecipeFor(RecipeType.CRAFTING, new RecipeWrapper(handler), level);
```

附加特性

Forge为配方纲要及其实现提供了一些额外的行为，以更好地控制系统。

配方的**ItemStack**结果

除了`minecraft:stonecutting`配方外，所有原版配方序列化器都会扩展`result`标签，以将完整的`ItemStack`作为`JsonObject`，而不是在某些情况下仅仅是物品名称和数量。

```
// 在某个配方JSON中
"result": {
  // 要作为结果提供的注册表物品的名称
  "item": "examplemod:example_item",
  // 要返回的物品数量
  "count": 4,
  // 物品栈的标签数据，也可以是一个字符串
  "nbt": {
      // 在此处添加标签数据
  }
}
```

注意

`nbt`标签也可以是一个字符串，其中包含无法正确表示为JSON对象（如`IntArrayTag`）的数据的字符串化NBT（或SNBT）。

条件性配方

配方及其所解锁的进度可以有条件地加载和保持默认，具体取决于存在的信息（模组的被加载、物品的存在等）。

更大的合成网格

默认情况下，原版声明合成网格的最大宽度和高度为3x3正方形。这可以通过在`FMLCommonSetupEvent`中使用新的宽度和高度调用`ShapedRecipe#setCraftingSize`来扩展。

警告

`ShapedRecipe#setCraftingSize`不是线程安全的。因此，它应该通过`FMLCommonSetupEvent#enqueueWork`排入同步工作队列。

配方中较大的合成网格可以是数据生成的。

原料类型

一些额外的原料类型被添加，以允许配方具有检查标签数据或将多种原料组合到单个输入检查器中的输入。

自定义配方

每个配方定义都由三个组件组成： `Recipe` 实现，它保存数据并用所提供的输入处理执行逻辑， `RecipeType` 表示配方将用于的类别或上下文，以及 `RecipeSerializer` ，它处理配方数据的解码和网络通信。如何选择使用配方取决于实施者。

配方

`Recipe` 接口描述配方数据和执行逻辑。这包括匹配输入并提供相关联的结果。由于配方子系统默认执行物品转换，因此输入是通过 `Container` 子类型提供的。

重要

传递到配方中的 `Container` 应被视为其内容是不可变的。任何可变操作都应该通过 `ItemStack#copy` 对输入的一份副本执行。

为了能够从管理器获得配方实例， `#matches` 必须返回true。此方法根据提供的容器进行检查，以查看相关联的输入是否有效。 `Ingredient` 可以通过调用 `Ingredient#test` 进行验证。

如果已经选择了配方，则使用 `#assemble` 构建配方，该 `#assemble` 可以使用来自输入的数据来创建结果。

提示

`#assemble` 应始终生成唯一的”`ItemStack` 。如果不确定 `#assemble` 是否执行此操作，请在返回之前对结果调用 `ItemStack#copy` 。

大多数其他方法纯粹是为了与配方相结合。

```
public record ExampleRecipe(Ingredient input, int data, ItemStack output) implements Recipe<Container> {
  // 在此处实现方法
}
```

注意

虽然在上面的示例中使用了一个记录，但在你自己的实现中不需要这样做。

RecipeType

`RecipeType` 负责定义配方将在其中使用的类别或上下文。例如，如果一个配方要在熔炉中熔炼，它的类型将是 `RecipeType#BLASTING` 。在高炉中进行熔炼的类型为 `RecipeType#BLASTING` 。

如果现有类型中没有一个与配方将在其中使用的上下文匹配，则必须注册一个新的 `RecipeType` 。

`RecipeType` 实例必须由新配方子类型中的 `Recipe#getType` 返回。

```
// 对于某个RegistryObject<RecipeType> EXAMPLE_TYPE
// 在ExampleRecipe中
@Override
public RecipeType<?> getType() {
  return EXAMPLE_TYPE.get();
}
```

RecipeSerializer

`RecipeSerializer` 负责解码JSON，并通过网络为关联的 `Recipe` 子类型进行通信。序列化器解码的每个配方都保存为 `RecipeManager` 中的唯一实例。 `RecipeSerializer` 必须已被注册。

`RecipeSerializer` 只需要实现三个方法：

方法	描述
fromJson	将JSON解码为 `Recipe` 子类型。
toNetwork	将 `Recipe` 编码到缓冲区以发送到客户端。配方标识符无需编码。
fromNetwork	从服务端发送的缓冲区中解码 `Recipe` 。配方标识符不需要解码。

然后，新配方子类型中的 `Recipe#getSerializer` 必须返回该 `RecipeSerializer` 实例。

```
// 对于某个RegistryObject<RecipeSerializer> EXAMPLE_SERIALIZER
// 在ExampleRecipe中
@Override
public RecipeSerializer<?> getSerializer() {
  return EXAMPLE_SERIALIZER.get();
}
```

> **提示**
>
> 有一些有用的方法可以让配方的读写数据变得更容易。 `Ingredient` 可以使用 `#fromJson` 、 `#toNetwork` 和 `#fromNetwork` ，而 `ItemStack` 可以使用 `CraftingHelper#getItemStack` 、 `FriendlyByteBuf#writeItem` 和 `FriendlyByteBuf#readItem` 。

构建JSON

自定义配方JSON与其他配方存储在同一个位置。指定的 `type` 应表示配方序列化器的注册表名称。任何附加数据都是由序列化器在解码期间指定的。

```
{
  // 自定义序列化器的注册表名称
  "type": "examplemod:example_serializer",
  "input": {
    // 某些原料输入
  },
  "data": 0, // 配方所需的一些数据
  "output": {
    // 某些物品栈输出
  }
}
```

非物品逻辑

如果物品未用作配方输入或结果的一部分，则 `RecipeManager` 中提供的常规方法将无效。相反，应将用于测试配方有效性和/或提供结果的附加方法添加到自定义 `Recipe` 实例中。从那里，特定 `RecipeType` 的所有配方都可以通过 `RecipeManager#getAllRecipesFor` 获得，然后使用新实现的方法进行检查和/或提供结果。

```
// 在某个Recipe子实现ExampleRecipe中
```

```
// 检查该位置的方块，看它是否与存储的数据匹配
boolean matches(Level level, BlockPos pos);

// 创建要将指定位置的方块设置为的方块状态
BlockState assemble(RegistryAccess access);

// 在某个管理器类中
public Optional<ExampleRecipe> getRecipeFor(Level level, BlockPos pos) {
  return level.getRecipeManager()
    .getAllRecipesFor(exampleRecipeType) // 获取所有配方
    .stream() // 在所有配方中查阅类型
    .filter(recipe -> recipe.matches(level, pos)) // 检查该配方输入是否合法
    .findFirst(); // 查找与输入匹配的第一个配方
}
```

数据生成

所有自定义配方，无论输入或输出数据如何，都可以使用 `RecipeProvider` 创建到用于数据生成的 `FinishedRecipe` 中。

原料

`Ingredient` 是基于物品的输入的predicate处理器，用于检查某个 `ItemStack` 是否满足成为配方中有效输入的条件。所有接受输入的原版配方都使用 `Ingredient` 或 `Ingredient` 的列表，然后将其合并为单一的 `Ingredient` 。

自定义原料

自定义原料可以通过将 `type` 设置为原料的序列化器的名称来指定，复合原料除外。当没有指定类型时， `type` 默认为原版原料 `minecraft:item` 。自定义原料也可以很容易地用于数据生成。

Forge类型

Forge提供了一些额外的 `Ingredient` 类型供程序员实现。

CompoundIngredient

尽管它们在功能上是相同的，但复合原料取代了在配方中实现原料列表的方式。它们作为一个逻辑或（OR）集合工作，其中传入的物品栈必须至少在一个提供的原料中。进行此更改是为了允许自定义原料在列表中正确工作。因此，无需指定类型。

```
// 对于某个输入
[
  // 这些原料中必须至少有一种必须匹配才能成功
  {
    // 原料
  },
  {
    // 自定义原料
    "type": "examplemod:example_ingredient"
  }
]
```

StrictNBTIngredient

`StrictNBTIngredient` 比较 `ItemStack` 上的物品、耐久和共享标签（由 `IForgeItem#getShareTag` 定义），以保证确切的等效性。这可以通过将 `type` 指定为 `forge:nbt` 来使用。

```
// 对于某个输入
{
  "type": "forge:nbt",
  "item": "examplemod:example_item",
  "nbt": {
    // 添加nbt数据（必须与物品栈上的数据完全匹配）
  }
}
```

PartialNBTIngredient

`PartialNBTIngredient` 是 `StrictNBTIngredient` 的宽松版本，因为它们与共享标签中指定的单个或一组物品以及仅键（由 `IForgeItem#getShareTag` 定义）进行比较。这可以通过将 `type` 指定为 `forge:partial_nbt` 来使用。

```
// 对于某个输入
{
  "type": "forge:partial_nbt",

  // 'item'或'items'必须被指定
  // 如果都指定了，那么只有'item'会被读取
  "item": "examplemod:example_item",
  "items": [
```

```
    "examplemod:example_item",
    "examplemod:example_item2"
    // ...
  ],

  "nbt": {
    // 仅检查'key1'和'key2'的等效性
    // 不会检查物品栈中的所有其他键
    "key1": "data1",
    "key2": {
      // 数据2
    }
  }
}
```

IntersectionIngredient

`IntersectionIngredient` 工作为一个逻辑和（AND）集合，其中传入的物品必须与所有提供的原料匹配。必须至少提供两种原料。这可以通过将 `type` 指定为 `forge:intersection` 来使用。

```
// 对于某个输入
{
  "type": "forge:intersection",

  // 所有这些原料都必须返回true才能成功
  "children": [
    {
      // 原料1
    },
    {
      // 原料2
    }
    // ...
  ]
}
```

DifferenceIngredient

`DifferenceIngredient` 工作为一个减法（SUB）集合，其中传入的物品栈必须与第一个原料匹配，但不能与第二个原料匹配。这可以通过将 `type` 指定为 `forge:difference` 来使用。

```
// 对于某个输入
{
  "type": "forge:difference",
  "base": {
    // 该物品栈所存在的原料
  },
  "subtracted": {
    // 该物品栈所不存在的原料
  }
}
```

创建自定义原料

可以通过为创建的 `Ingredient` 子类实现 `IIngredientSerializer` 来创建自定义原料。

提示

自定义原料应该是`AbstractIngredient`的子类，因为它提供了一些有用的抽象以便于实现。

原料的子类

对于每个原料子类，有三种重要的方法需要实现：

方法	描述
getSerializer	返回用于读取和写入原料的serializer。
test	如果输入对此原料有效，则返回true。
isSimple	如果原料与物品栈的标签匹配，则返回false。`AbstractIngredient`的子类需要定义此行为，而`Ingredient`子类默认返回true。

所有其他定义的方法都留给读者练习，以便根据原料子类的需要使用。

IIngredientSerializer

`IIngredientSerializer`子类型必须实现三种方法：

方法	描述
parse (JSON)	将`JsonObject`转换为`Ingredient`。
parse (Network)	返回用于解码`Ingredient`的网络缓冲区。
write	将一个`Ingredient`写入网络缓冲区。

此外，`Ingredient`子类应实现`Ingredient#toJson`，以便与数据生成一起使用。`AbstractIngredient`的子类使`#toJson`成为一个需要实现该方法的抽象方法。

之后，应声明一个静态实例来保存初始化的序列化器，然后在`RecipeSerializer`的`RegisterEvent`期间或在`FMLCommonSetupEvent`期间使用`CraftingHelper#register`进行注册。`Ingredient`子类在`Ingredient#getSerializer`中返回序列化器的静态实例。

```
// 在某个序列化器类中
public static final ExampleIngredientSerializer INSTANCE = new ExampleIngredientSerializer();

// 在某个处理器类中
public void registerSerializers(RegisterEvent event) {
  event.register(ForgeRegistries.Keys.RECIPE_SERIALIZERS,
    helper -> CraftingHelper.register(registryName, INSTANCE)
  );
}

// 在某个原料类中
@Override
public IIngredientSerializer<? extends Ingredient> getSerializer() {
  return INSTANCE;
}
```

提示

如果使用`FMLCommonSetupEvent`注册原料序列化器，则必须通过`FMLCommonSetupEvent#enqueueWork`将其排入同步工作队列，因为`CraftingHelper#register`不是线程安全的。

非数据包配方

并不是所有的配方都足够简单或迁移到使用数据驱动的配方。一些子系统仍然需要在代码库中进行修补，以提供对添加新配方的支持。

酿造配方

酿造是代码中为数不多的仍然存在的配方之一。酿造配方是作为 `PotionBrewing` 中的引导程序的一部分添加的，用于容器、容器配方和药水混合物。为了扩展现有系统，Forge允许通过在 `FMLCommonSetupEvent` 中调用 `BrewingRecipeRegistry#addRecipe` 来添加酿造配方。

> **警告**
>
> `BrewingRecipeRegistry#addRecipe` 必须在同步工作队列中通过 `#enqueueWork` 调用，因为该方法不是线程安全的。

默认实现接受标准实现的输入成分、催化剂成分和物品栈输出。此外，还可以提供一个 `IBrewingRecipe` 实例来执行转换。

IBrewingRecipe

`IBrewingRecipe` 是一个伪 `Recipe` 接口，用于检查输入和催化剂是否有效，并在有效时提供相关输出。它分别通过 `#isInput` 、 `#isIngredient` 和 `#getOutput` 提供。输出方法可以访问输入和催化剂物品栈来构建结果。

> **重要**
>
> 在 `ItemStack` 或 `CompoundTag` 之间复制数据时，请确保使用它们各自的 `#copy` 方法来创建唯一的实例。

没有类似原版的包装来添加额外的药水容器或药水混合物。需要添加一个新的 `IBrewingRecipe` 实现来复制此行为。

铁砧配方

铁砧负责接收损坏的输入，并给定一些材料或类似的输入，消除输入结果上的一些损坏。因此，它的系统不是简易地被数据驱动。然而，当用户具有所需的经验等级时，由于铁砧配方是具有一定数量的材料等于一定输出的输入，因此可以通过 `AnvilUpdateEvent` 对其进行修改以创建伪配方系统。这接受了输入和材料，并允许模组开发者指定输出、经验等级成本和用于输出的材料数量。该事件还可以通过取消来阻止任何输出。

```
// 检查左边和右边的物品是否正确
// 当正确时，设置输出，经验等级消耗，以及材料数量
public void updateAnvil(AnvilUpdateEvent event) {
  if (event.getLeft().is(...) && event.getRight().is(...)) {
    event.setOutput(...);
    event.setCost(...);
    event.setMaterialCost(...);
  }
}
```

该更新事件必须被绑定到Forge事件总线。

织布机配方

织布机负责将染料和图案（从织布机或物品上）应用到旗帜上。虽然旗帜和染料必须分别为 `BannerItem` 或 `DyeItem` ，但可以在织布机中创建和应用自定义图案。旗帜图案可以通过注册一个 `BannerPattern` 来创建。

重要

`minecraft:no_item_required` 标签中的 `BannerPattern` 在织布机中作为一个选项出现。不在此标签中的图案必须有一个附带的 `BannerPatternItem` 才能与关联的标签一起使用。

```
private static final DeferredRegister<BannerPattern> REGISTER = DeferredRegister.create(Registries.BANNER_PATTERN, "exa

// 接受要通过网络发送的图案名称
public static final BannerPattern EXAMPLE_PATTERN = REGISTER.register("example_pattern", () -> new BannerPattern("examp
```

13.2.3 战利品表

战利品表是逻辑文件，它规定了当发生各种操作或场景时应该发生什么。尽管原版系统纯粹处理物品生成，但该系统可以扩展为执行任意数量的预定义操作。

由数据驱动的表

原版中的大多数战利品表都是通过JSON进行数据驱动的。这意味着模组不需要创建新的战利品表，只需要数据包。关于如何在模组的 `resources` 文件夹中创建和放置这些战利品表的完整列表可以在Minecraft Wiki上找到。

使用战利品表

战利品表由其指向 `data/<namespace>/loot_tables/<path>.json` 的 `ResourceLocation` 引用。与引用相关联的 `LootTable` 可以使用 `LootTables#get` 获得，其中 `LootTables` 可以通过 `MinecraftServer#getLootTables` 获得。

战利品表总是在给定的上下文中生成的。 `LootContext` 定义了表的生成存档、特定的随机化器和种子（如果需要）、更好生成的运气、定义场景上下文的 `LootContextParam` 以及激活时应出现的任何动态信息。可以使用 `LootContext$Builder` 的构造函数创建战利品上下文，并使用 `LootContext$Builder#create` 构建战利品上下文。

创建的 `LootContext` 遵循某些 `LootContextParamSet` 。参数集定义在上下文中哪些 `LootContextParam` 是必需的或可选的，以便生成。在给定参数集中生成的战利品表必须仅使用已定义的上下文。

`LootTable` 可用于使用以下可用方法之一生成 `ItemStack` ：

方法	描述
`getRandomItemsRaw`	消耗由战利品表生成的物品。
`getRandomItems`	返回由战利品表生成的物品。
`fill`	用已生成的战利品表填充容器。

> **注意**
>
> 战利品表是为生成物品而构建的，因此这些方法需要对 `ItemStack` 进行一些处理。

附加特性

Forge为战利品表提供了一些额外的行为，以更好地控制系统。

`LootTableLoadEvent`

`LootTableLoadEvent` 是在Forge事件总线上触发的事件，每当加载战利品表时就会触发。如果事件被取消，则会加载一个空的战利品表。

> **重要**
>
> 不要通过此事件修改战利品表的掉落。这些修改应该使用全局战利品修改器来完成。

战利品池名称

Loot pools can be named using the `name` key. Any non-named loot pool will be the hash code of the pool prefixed by `custom#` . 可以使用 `name` 键对战利品池进行命名。任何未命名的战利品池都将是以 `custom#` 为前缀的池的哈希代码。

```
// 对于某个战利品池
{
  "name": "example_pool", // 战利品池将被命名为'example_pool'
  "rolls": {
    // ...
  },
  "entries": {
    // ...
  }
}
```

抢夺修改器

战利品表现在除了受到抢夺附魔的影响外，还受到Forge事件总线上的 `LootingLevelEvent` 的影响。

附加的上下文参数

Forge扩展了某些参数集，以解决可能适用的缺失上下文。 `LootContextParamSets#CHEST` 现在允许使用 `LootContextParams#KILLER_ENTITY` ，因为箱子矿车是可以被破坏（或“杀死”）的实体。 `LootContextParamSets#FISHING` 还允许 `LootContextParams#KILLER_ENTITY` ，因为鱼钩也是一个实体，当玩家取回它时会收回（或“杀死”）。

熔炼时的多个物品

当使用 `SmeltItemFunction` 时，熔炼配方现在将返回结果中的实际物品数，而不是单个熔炼物品（例如，如果熔炼配方返回3个物品，并且有3次掉落，则结果将是9个熔炼物品，而不是3个）。

战利品表**Id**条件

Forge添加了一个额外的 `LootItemCondition` ，允许为特定的表生成某些物品。这通常用于全局战利品修改器。

```
// 在某个战利品池或池条目中
{
  "conditions": [
    {
      "condition": "forge:loot_table_id",
      // 当该战利品表对于泥土时将适用
      "loot_table_id": "minecraft:blocks/dirt"
    }
  ]
}
```

“工具能否执行操作”条件

Forge添加了一个额外的 `LootItemCondition` ，用于检查给定的 `LootContextParams#TOOL` 是否可以执行指定的 `ToolAction` 。

```
// 在某个战利品池或池条目中
{
  "conditions": [
    {
      "condition": "forge:can_tool_perform_action",
      // 当该工具可以像斧一样剥下原木时将适用
      "action": "axe_strip"
    }
```

```
  ]
}
```

13.2.4 全局战利品修改器

全局战利品修改器是一种数据驱动的方法，可以处理收割掉落的修改，而无需覆盖数十到数百个原版战利品表，也无需处理需要与另一个模组的战利品表交互的效果，而不知道可能加载了什么模组。全局战利品修改器也是堆叠的，而不是后来者为王，类似于标签。

注册一个全局战利品修改器

你将需要4件事物：

1. 创建一个 `global_loot_modifiers.json` 。
    - 这将告诉Forge你的修改器以及类似于tags[标签]的工作。
2. 代表修改器的序列化json。
    - 这将包含有关你修改的所有数据，并允许数据包调整你的效果。
3. 一个继承自 `IGlobalLootModifier` 的类。
    - 使修改器工作的操作代码。大多数模组开发者都可以继承 `LootModifier` ，因为它提供了基本功能。
4. 最后，使用编解码器对操作类进行编码和解码。
    - 其应像任何其他 `IForgeRegistryEntry` 一样被注册。

`global loot modifiers.json` 文件

`global_loot_modifiers.json` 表示要加载到游戏中的所有战利品修改器。此文件必须放在 `data/forge/loot_modifiers/global_loot_modifiers.json` 。

> **重要**
>
> `global_loot_modifiers.json` 只能在 `forge` 命名空间中被读取。如果该文件位于模组的命名空间下，则会被忽略。

`entries` 是将要加载的修改器的有序列表。指定的ResourceLocation指向其在 `data/<namespace>/loot_modifiers/<path>.json` 中的关联条目。这主要与数据包生成器有关，用于解决独立模组的修改器之间的冲突。

`replace` ，当 `true` 时，会将行为从向全局列表添加战利品修改器更改为完全替换全局列表条目。为了与其他模组实现兼容，模组开发者将希望使用 `false` 。数据包作者可能希望用 `true` 以指定其覆盖。

```
{
  "replace": false, // 必须存在
  "entries": [
    // 代表'data/examplemod/loot_modifiers/example_glm.json'中的一个战利品修改器
    "examplemod:example_glm",
    "examplemod:example_glm2"
    // ...
  ]
}
```

序列化**JSON**

该文件包含与修改器相关的所有潜在变量，包括修改任何战利品之前必须满足的条件。尽可能避免硬编码值，以便数据包作者可以根据需要调整平衡。

`type` 表示用于读取关联JSON文件的编解码器的注册表名称。这必须始终存在。

`conditions` 应该表示该修改器要激活的战利品表条件。条件应该避免被硬编码，以允许数据包作者尽可能灵活地调整标准。这也必须始终存在。

重要

尽管 `conditions` 应该表示修改器激活所需的内容，但只有在使用捆绑的Forge类时才会出现这种情况。如果使用 `LootModifier` 作为子类，则所有条件都将用逻辑与（**AND**）相连，并检查是否应应用修改器。

还可以指定由序列化器读取并由修改器定义的任何附加属性。

```
// 在data/examplemod/loot_modifiers/example_glm.json内
{
  "type": "examplemod:example_loot_modifier",
  "conditions": [
    // 普通的战利品表条件
    // ...
  ],
  "prop1": "val1",
  "prop2": 10,
  "prop3": "minecraft:dirt"
}
```

IGlobalLootModifier

要提供全局战利品修改器指定的功能，必须指定一个 `IGlobalLootModifier` 实现。这些是每次序列化器解码JSON中的信息并将其提供给该对象时生成的实例。

为了创建新的修改器，需要定义两种方法： `#apply` 和 `#codec` 。 `#apply` 获取将与上下文信息一起生成的当前战利品，例如当前等级或额外定义的参数。它返回要生成的掉落物列表。

注意

从任何一个修改器返回的掉落物列表都会按照它们注册的顺序输入到其他修改器中。因此，修改后的战利品可以被另一个战利品修改器修改。

`#codec` 返回已注册的编解码器，用于将修改器编码到JSON或从JSON解码修改器。

LootModifier 子类

`LootModifier` 是 `IGlobalLootModifier` 的一个抽象实现，用于提供大多数模组开发者可以轻松扩展和实现的基本功能。其通过定义 `#apply` 方法来检查条件，以确定是否修改生成的战利品，从而扩展了现有接口。

在子类实现中有两件事需要注意：构造函数必须接受 `LootItemCondition` 的一个数组和 `#doApply` 方法。

`LootItemCondition` 的数组定义了在修改战利品之前必须为true的条件列表。所提供的条件是用逻辑和（**AND**）连在一起的，这意味着所有条件都必须为true。

`#doApply` 方法的工作原理与 `#apply` 方法相同，只是它只在所有条件都返回true时执行。

```
public class ExampleModifier extends LootModifier {

  public ExampleModifier(LootItemCondition[] conditionsIn, String prop1, int prop2, Item prop3) {
    super(conditionsIn);
    // 存储其余参数
  }

  @NotNull
  @Override
  protected ObjectArrayList<ItemStack> doApply(ObjectArrayList<ItemStack> generatedLoot, LootContext context) {
    // 修改战利品并返回新的掉落物
```

```
  }

  @Override
  public Codec<? extends IGlobalLootModifier> codec() {
    // 返回用于编码和解码此修改器的编解码器
  }
}
```

战利品修改器的编解码器

JSON和 `IGlobalLootModifier` 实例之间的桥梁是 `Codec<T>`，其中 `T` 表示要使用的 `IGlobalLootModifier` 的具体类型。

为了方便起见，通过 `LootModifier#codecStart` 为类似记录的编解码器提供了一个战利品条件编解码器。这用于相关战利品修改器的数据生成。

```
// 对于某个DeferredRegister<Codec<? extends IGlobalLootModifier>> REGISTRAR
public static final RegistryObject<Codec<ExampleModifier>> = REGISTRAR.register("example_codec", () ->
  RecordCodecBuilder.create(
    inst -> LootModifier.codecStart(inst).and(
      inst.group(
        Codec.STRING.fieldOf("prop1").forGetter(m -> m.prop1),
        Codec.INT.fieldOf("prop2").forGetter(m -> m.prop2),
        ForgeRegistries.ITEMS.getCodec().fieldOf("prop3").forGetter(m -> m.prop3)
      )
    ).apply(inst, ExampleModifier::new)
  )
);
```

示例可以在Forge Git存储库中找到，包括精准采集和熔炼效果。

13.2.5 标签

标签是游戏中用于将相关事物分组在一起并提供快速成员身份检查的通用对象集。

声明你自己的组别

标签在你的模组的数据包中声明。例如，给定标识符为 `modid:foo/tagname` 的 `TagKey<Block>` 将引用位于 `/data/<modid>/tags/blocks/foo/tagname.json` 的标签。 `Block` 、 `Item` 、 `EntityType` 、 `Fluid` 以及 `GameEvent` 的标签，将使用复数形式作为其文件夹位置，而所有其他注册表使用单数形式（ `EntityType` 使用文件夹 `entity_types` ，而 `Potion` 将使用文件夹 `potion` ）。 类似地，你可以通过声明自己的JSON来附加或覆盖在其他域（如原版）中声明的标签。 例如，要将你自己的模组的树苗添加到原版树苗标签中，你可以在 `/data/minecraft/tags/blocks/saplings.json` 中指定它，如果 `replace` 选项为false，原版将在重载时将所有内容合并到一个标签中。 如果 `replace` 为true，那么指定 `replace` 的json之前的所有条目都将被删除。 列出的不存在的值将导致标签出错，除非使用 `id` 字符串和设置为false的 `required` 布尔值列出该值，如以下示例所示：

```
{
  "replace": false,
  "values": [
    "minecraft:gold_ingot",
    "mymod:my_ingot",
    {
      "id": "othermod:ingot_other",
      "required": false
    }
  ]
}
```

有关基本语法的描述，请参阅原版wiki。

原版语法上还有一个Forge扩展。 你可以声明一个与 `values` 数组格式相同的 `remove` 数组。此处列出的任何值都将从标签中删除。这相当于原版 `replace` 选项的细粒度版本。

在代码中使用标签

登录和重新加载时，所有注册表的标签都会自动从服务器发送到任何远程客户端。 `Block` 、 `Item` 、 `EntityType` 、 `Fluid` 和 `GameEvent` 都被特殊地包装，因为它们具有 `Holder` ，允许通过对象本身访问可用标签。

注意

在未来版本的Minecraft中，侵入性的 `Holder` 可能会被移除。如果被移除了，则可以使用以下方法来查询关联的 `Holder` 。

ITagManager

Forge封装的注册表提供了一个额外的帮助，用于通过 `ITagManager` 创建和管理标签，该标签可以通过 `IForgeRegistry#tags` 获得。可以使用 `#createTagKey` 或 `#createOptionalTagKey` 创建标签。标签或注册表对象也可以分别使用 `#getTag` 或 `#getReverseTag` 检查。

自定义注册表

自定义注册表可以在分别通过 `#createTagKey` 或 `#createOptionalTagKey` 构造其 `DeferredRegister` 时创建标签。然后，可以使用通过调用 `DeferredRegister#makeRegistry` 获得的 `IForgeRegistry` 来检查它们的标签或注册表对象。

引用标签

创建标签包装有四种方法：

方法	对于
`*Tags#create`	`BannerPattern` 、 `Biome` 、 `Block` 、 `CatVariant` 、 `DamageType` 、 `EntityType` 、 `FlatLevelGeneratorPreset` 、 `GameEvent` 、 `Item` 、 `PaintingVariant` 、 `PoiType` 、 `StructureFeature` 、 `StructureProcessorList` 、 `StructureSet` 、 `TemplatePool` 及 `WorldPreset` ，其中 `*` 代表这些类型之一。
`ITagManager#createTagKey`	由Forge包装的原版注册表，可从 `ForgeRegistries` 取得。
`DeferredRegister#createTagKey`	自定义的Forge注册表。
`TagKey#create`	无Forge包装的原版注册表，可从 `Registry` 取得。

注册表对象可以通过其 `Holder` 或通过 `ITag` / `IReverseTag` 分别检查其标签或注册表对象是否为原版或Forge注册表对象。

原版注册表对象可以使用 `Registry#getHolder` 或 `Registry#getHolderOrThrow` 获取其关联的持有者，然后使用 `Holder#is` 比较注册表对象是否有标签。

Forge注册表对象可以使用 `ITagManager#getTag` 或 `ITagManager#getReverseTag` 获取其标签定义，然后分别使用 `ITag#contains` 或 `IReverseTag#containsTag` 比较注册表对象是否有标签。

持有标签的注册表对象在其注册表对象或状态感知类中包含一个名为 `#is` 的方法，以检查该对象是否属于某个标签。

举一个例子：

```
public static final TagKey<Item> myItemTag = ItemTags.create(new ResourceLocation("mymod", "myitemgroup"));

public static final TagKey<Potion> myPotionTag = ForgeRegistries.POTIONS.tags().createTagKey(new ResourceLocation("mymod", "my

public static final TagKey<VillagerType> myVillagerTypeTag = TagKey.create(Registries.VILLAGER_TYPE, new ResourceLocation("myr

// 在某个方法中：

ItemStack stack = /*...*/;
boolean isInItemGroup = stack.is(myItemTag);

Potion potion = /*...*/;
boolean isInPotionGroup  = ForgeRegistries.POTIONS.tags().getTag(myPotionTag).contains(potion);

ResourceKey<VillagerType> villagerTypeKey = /*...*/;
boolean isInVillagerTypeGroup = BuiltInRegistries.VILLAGER_TYPE.getHolder(villagerTypeKey).map(holder -> holder.is(myVillager
```

惯例

有几个惯例将有助于促进该生态系统中的兼容性：

- 如果有适合你的方块或物品的原版标签，请将其添加到该标签中。请参阅原版标签列表。
- 如果有一个Forge标签适合你的方块或物品，请将其添加到该标签中。Forge声明的标签列表可以在GitHub上看到。
- 如果有一组你认为应该由社区共享的东西，请使用 `forge` 命名空间，而不是你的mod id。
- 标签命名约定应遵循原版约定。特别是，物品和方块组别是复数而不是单数（例如 `minecraft:logs` 、 `minecraft:saplings` ）。
- 物品标签应根据其类型分类到子目录中（例如 `forge:ingots/iron` 、 `forge:nuggets/brass` 等）。

从**OreDictionary**迁移

- 对于配方，标签可以直接以原版配方格式使用（见下文）。
- 有关代码中的匹配物品，请参阅上面的章节。
- 如果你要声明一种新类型的物品组别，请遵循以下几个命名约定：
- 使用 `domain:type/material` 。当名称是所有模组开发者都应该采用的通用名称时，请使用 `forge` 域。
- 例如，铜锭应在 `forge:ingots/brass` 标签下注册，钴粒应在 `forge:nuggets/cobalt` 标签下注册。

在配方和进度中使用标签

原版直接支持标签。有关用法的详细信息，请参阅配方和进度的原版wiki页面。

13.2.6 进度

进度是玩家可以实现的任务，可以推进游戏的进度。进度可以基于玩家可能直接参与的任何动作来触发。

原版中的所有进度实现都是通过JSON进行数据驱动的。这意味着模组不需要创建新的进度，只需要数据包。关于如何创建这些进度并将其放入模组的 `resources` 中的完整列表可以在Minecraft Wiki上找到。此外，进度可以有条件加载和或保持默认，这取决于存在的信息（模组被加载、物品的存在等）。

进度标准

若要解锁一个进度，必须满足指定的标准。通过执行某个动作时执行的触发器来跟踪标准：杀死实体、更改物品栏、给动物喂食等。任何时候将进度加载到游戏中，定义的标准都会被读取并添加为触发器的监听器。然后调用一个触发器函数（通常称为 `#trigger` ），该函数检查所有监听器当前状态是否满足进度标准的条件。只有在通过完成所有条件获得进度后，才会删除进度的标准监听器。

需求被定义为包含字符串数组的一个数组，该数组表示在进度中指定的标准的名称。一旦满足一个字符串数组的条件，就完成了进度：

```
// 在某个进度JSON中

// 所定义的要满足的标准的列表
"criteria": {
  "example_criterion1": { /*...*/ },
  "example_criterion2": { /*...*/ },
  "example_criterion3": { /*...*/ },
  "example_criterion4": { /*...*/ }
},

// 该进度只能解锁一次
// - 标准1和2均被满足
// 或
// - 标准3和4均被满足
"requirements": [
  [
    "example_criterion1",
    "example_criterion2"
  ],
  [
    "example_criterion3",
    "example_criterion4"
  ]
]
```

原版定义的标准触发器列表可以在 `CriteriaTriggers` 中找到。此外，JSON格式是在Minecraft Wiki上定义的。

自定义标准触发器

可以通过为已创建的 `AbstractCriterionTriggerInstance` 子类实现 `SimpleCriterionTrigger` 来创建自定义条件触发器。

AbstractCriterionTriggerInstance子类

`AbstractCriterionTriggerInstance` 表示在 `criteria` 对象中定义的单个标准。触发器实例负责保存定义的条件，返回输入是否与条件匹配，并将实例写入JSON用于数据生成。

条件通常通过构造函数传递。`AbstractCriterionTriggerInstance` 父级构造函数要求实例将触发器的注册表名和玩家必须满足的条件定义为 `EntityPredicate$Composite` 。触发器的注册表名称应该直接提供给父级，而玩家的条件应该是构造函数参数。

```
// 其中ID是该触发器的注册表名称
public ExampleTriggerInstance(EntityPredicate.Composite player, ItemPredicate item) {
  super(ID, player);
  // 存储必须满足的物品条件
}
```

注意

通常，触发器实例有一个静态构造函数，允许轻松创建这些实例以生成数据。这些静态工厂方法也可以静态导入，而不是类本身。

```
public static ExampleTriggerInstance instance(EntityPredicate.Builder playerBuilder, ItemPredicate.Builder itemBuilder)
  return new ExampleTriggerInstance(EntityPredicate.Composite.wrap(playerBuilder.build()), itemBuilder.build());
}
```

此外，应该重写 `#serializeToJson` 方法。该方法应该将实例的条件添加到其他JSON数据中。

```
@Override
public JsonObject serializeToJson(SerializationContext context) {
  JsonObject obj = super.serializeToJson(context);
  // 将条件写入json中
  return obj;
}
```

最后，应该添加一个方法，该方法接受当前数据状态并返回用户是否满足必要条件。玩家的条件已经通过 `SimpleCriterionTrigger#trigger(ServerPlayer, Predicate)` 进行了检查。大多数触发器实例称这个方法为 `#matches` 。

```
// 此方法对于每个实例都是唯一的，因此不会被重写
public boolean matches(ItemStack stack) {
  // 由于ItemPredicate与一个物品栈匹配，因此一个物品栈是输入
  return this.item.matches(stack);
}
```

SimpleCriterionTrigger

`SimpleCriterionTrigger<T>` 子类，其中 `T` 是触发器实例的类型，负责指定触发器的注册表名、创建触发器实例以及检查触发器实例和在成功时运行附加监听器的方法。

触发器的注册表名称被提供给 `#getId` 。这应该与提供给触发器实例的注册表名称相匹配。

触发器实例是通过 `#createInstance` 创建的。此方法从JSON中读取一个标准。

```
@Override
public ExampleTriggerInstance createInstance(JsonObject json, EntityPredicate.Composite player, DeserializationContext context
  // 从JSON中读取条件：item
  return new ExampleTriggerInstance(player, item);
}
```

最后，定义了一个方法来检查所有触发器实例，并在满足它们的条件时运行监听器。此方法接受 `ServerPlayer` 和 `AbstractCriterionTriggerInstance` 子类中匹配的方法定义的任何其他数据。此方法应在内部调用 `SimpleCriterionTrigger#trigger` 以正确处理检查所有监听器。大多数触发器实例从称这个方法为 `#trigger` 。

```
// 此方法对于每个触发器都是唯一的，因此不会被重写
public void trigger(ServerPlayer player, ItemStack stack) {
  this.trigger(player,
    // AbstractCriterionTriggerInstance子类中的条件检查器方法
    triggerInstance -> triggerInstance.matches(stack)
  );
}
```

之后，应在 `FMLCommonSetupEvent` 期间使用 `CriteriaTriggers#register` 注册实例。

> **重要**
>
> `CriteriaTriggers#register` 必须通过 `FMLCommonSetupEvent#enqueueWork` 排入同步工作队列，因为该方法不是线程安全的。

触发器的调用

每当执行被检查的操作时，都应该调用 `SimpleCriterionTrigger` 子类定义的 `#trigger` 方法。

```
// 在执行操作的某段代码中
// 其中EXAMPLE_CRITERIA_TRIGGER是自定义标准触发器
public void performExampleAction(ServerPlayer player, ItemStack stack) {
  // 运行代码以执行操作
  EXAMPLE_CRITERIA_TRIGGER.trigger(player, stack);
}
```

进度奖励

当进度达成时，可以给予奖励。其可以是经验点数、战利品表、配方书的配方的组合，也可以是作为创造模式玩家执行的函数。

```
// In some advancement JSON
"rewards": {
  "experience": 10,
  "loot": [
    "minecraft:example_loot_table",
    "minecraft:example_loot_table2"
    // ...
  ],
  "recipes": [
    "minecraft:example_recipe",
    "minecraft:example_recipe2"
    // ...
  ],
  "function": "minecraft:example_function"
}
```

13.2.7 条件性加载数据

有时，模组开发者可能希望包括一些使用来自另一个模组的信息的数据驱动的对象，而不必明确地使该模组成为依赖项。其他情况可能是，当某些对象存在时，将其与其他模组编写的条目交换。这可以通过条件子系统来完成。

实现

目前，条件加载已针对配方和进度实现。对于任何有条件的配方或进度，都会加载一个条件到数据对的列表。如果为列表中的某个数据指定的条件为true，则返回该数据。否则，将丢弃该数据。

```
{
  // 需要为配方指定类型，因为它们可以具有自定义序列化器
  // 进度不需要这种类型
  "type": "forge:conditional",

  "recipes": [ // 或'advancements'（对于进度）
    {
      // 要检查的条件
      "conditions": [
        // 该列表中的条件用逻辑和（AND）相连
        {
          // 条件1
        },
        {
          // 条件2
        }
      ],
      "recipe": { // 或'advancement'（对于进度）
        // 如果所有条件都成功，则使用的配方
      }
    },
    {
      // 如果上一个条件失败，则接下来要检查的条件
    },
  ]
}
```

通过 `ConditionalRecipe$Builder` 和 `ConditionalAdvancement$Builder` ，条件加载的数据还具有用于数据生成的包装。

条件

条件是通过将 `type` 设置为 `IConditionSerializer#getID` 指定的条件名称来指定的。

True和False

布尔条件不包含任何数据，并返回条件的期望值。它们用 `forge:true` 和 `forge:false` 来表示。

```
// 对于某个条件
{
  // 将始终返回true（或为'forge:false'时始终返回false）
  "type": "forge:true"
}
```

Not、And和Or

布尔运算符条件由正在操作的条件组成，并应用以下逻辑。它们用 `forge:not` 、 `forge:and` 和 `forge:or` 表示。

```
// 对于某个条件
{
  // 反转存储条件的结果
  "type": "forge:not",
  "value": {
    // 一个条件
  }
}
```

```
// 对于某个条件
{
  // 将存储条件用逻辑和（AND）相连（或为'forge:or'时将存储条件用逻辑或（OR）相连）
  "type": "forge:and",
  "values": [
    {
      // 第一个条件
    },
    {
      // 第二个要用逻辑和（AND）连接的条件（或为'forge:or'时用逻辑或（OR）连接）
    }
  ]
}
```

模组被加载

只要在当前应用程序中加载了具有给定id的指定模组， `ModLoadedCondition` 就会返回true。其由 `forge:mod_loaded` 表示。

```
// 对于某个条件
{
  "type": "forge:mod_loaded",
   // 如果'examplemod'已被加载，则返回true
  "modid": "examplemod"
}
```

物品存在

只要给定物品已在当前应用程序中注册， `ItemExistsCondition` 就会返回true。其由 `forge:item_exists` 表示。

```
// 对于某个条件
{
  "type": "forge:item_exists",
   // 如果'examplemod:example_item'已被注册，则返回true
  "item": "examplemod:example_item"
}
```

标签为空

只要给定的物品标签中没有物品， `TagEmptyCondition` 就会返回true。其由 `forge:tag_empty` 表示。

```
// 对于某个条件
{
  "type": "forge:tag_empty",
   // 如果'examplemod:example_tag'是一个没有条目的物品标签，则返回true
```

```
  "tag": "examplemod:example_tag"
}
```

创建自定义条件

可以通过实现 `ICondition` 及与其关联的 `IConditionSerializer` 来创建自定义条件。

ICondition

任何条件只需要实现两种方法：

方法	描述
getID	该条件的注册表名称。必须等效于 `IConditionSerializer#getID` 。仅用于数据生成。
test	当条件满足时返回true。

注意

每个 `#test` 都可以访问一些代表游戏状态的 `IContext` 。目前，从注册表中只能获取标签。

IConditionSerializer

序列化器需要实现三种方法：

方法	描述
getID	该条件的注册表名称。必须等效于 `ICondition#getID` 。
read	从JSON中读取条件数据。
write	将给定的条件数据写入JSON。

注意

条件序列化器不负责写入或读取序列化器的类型，类似于Minecraft中的其他序列化器实现。

之后，应声明一个静态实例来保存初始化的序列化器，然后在 `RecipeSerializer` 的 `RegisterEvent` 期间或在 `FMLCommonSetupEvent` 期间使用 `CraftingHelper#register` 进行注册。

```
// 在某个序列化器类中
public static final ExampleConditionSerializer INSTANCE = new ExampleConditionSerializer();

// 在某个处理器类中
public void registerSerializers(RegisterEvent event) {
  event.register(ForgeRegistries.Keys.RECIPE_SERIALIZERS,
    helper -> CraftingHelper.register(INSTANCE)
  );
}
```

重要

如果使用`FMLCommonSetupEvent`注册条件序列化器，则必须通过`FMLCommonSetupEvent#enqueueWork`将其排入同步工作队列，因为`CraftingHelper#register`不是线程安全的。

14. 数据生成

14.1 数据生成

数据生成器是以编程方式生成模组的资源（asset）和数据（data）的一种方式。它允许在代码中定义这些文件的内容并自动生成它们，而不必担心细节。

数据生成器系统由主类 `net.minecraft.data.Main` 加载。可以传递不同的命令行参数来自定义收集了哪些模组的数据，考虑了哪些现有文件等。负责数据生成的类是 `net.minecraft.data.DataGenerator` 。

MDK的 `build.gradle` 中的默认配置添加了用于运行数据生成器的 `runData` 任务。

14.1.1 现存的文件

对未为数据生成而生成的纹理或其他数据文件的所有引用都必须引用系统上的现有文件。这是为了确保所有引用的纹理都在正确的位置，这样就可以找到并更正拼写错误。

`ExistingFileHelper` 是负责验证这些数据文件是否存在的类。可以从 `GatherDataEvent#getExistingFileHelper` 中检索实例。

`--existing <folderpath>` 参数允许在验证文件是否存在时使用指定的文件夹及其子文件夹。此外， `--existing-mod <modid>` 参数允许将加载的模组的资源用于验证。默认情况下，只有普通的数据包和资源可用于 `ExistingFileHelper` 。

14.1.2 生成器模式

数据生成器可以配置为运行4个不同的数据生成，这些数据生成是通过命令行参数配置的，并且可以通过 `GatherDataEvent#include***` 方法进行检查。

- **Client Assets**
- 在 `assets` 中生成仅客户端文件：f方块/物品模型、方块状态JSON、语言文件等。
- `--client` , `#includeClient`
- **Server Data**
- 在 `data` 中生成仅服务端文件：配方、进度、标签等。
- `--server` , `#includeServer`
- **Development Tools**
- 运行一些开发工具：将SNBT转换为NBT，反之亦然，等等。
- `--dev` , `#includeDev`
- **Reports**
- 转储所有已注册的方块、物品、命令等。
- `--reports` , `#includeReports`

所有的生成器都可以使用 `--all` 包含在内。

14.1.3 数据提供者

数据提供者是实际定义将生成和提供哪些数据的类。所有数据提供者都实现 `DataProvider` 。Minecraft对大多数asset和data都有抽象实现，因此模组开发者只需要扩展和覆盖指定的方法。

当创建数据生成器时，在模组事件总线上触发 `GatherDataEvent` ，并且可以从事件中获取 `DataGenerator` 。使用 `DataGenerator#addProvider` 创建和注册数据提供者。

客户端资源（**Assets**）

- `net.minecraftforge.common.data.LanguageProvider` - 针对语言设置；实现 `#addTranslations`
- `net.minecraftforge.common.data.SoundDefinitionsProvider` - 针对 `sounds.json` ；实现 `#registerSounds`
- `net.minecraftforge.client.model.generators.ModelProvider<?>` - 针对[模型]；实现 `#registerModels`
  - `ItemModelProvider` - 针对物品模型
  - `BlockModelProvider` - 针对方块模型
- `net.minecraftforge.client.model.generators.BlockStateProvider` - 针对方块状态JSON以及其方块和物品模型；实现 `#registerStatesAndModels`

服务端数据（**Data**）

这些类在 `net.minecraftforge.common.data` 包之下：

- `GlobalLootModifierProvider` - 针对全局战利品修改器；实现 `#start`
- `DatapackBuiltinEntriesProvider` - 针对数据包注册表对象；向构造函数传递 `RegistrySetBuilder`

这些类在 `net.minecraft.data` 包之下：

- `loot.LootTableProvider` - 针对战利品表；向构造函数传递 `LootTableProvider$SubProviderEntry`
- `recipes.RecipeProvider` - 针对[配方]以及其解锁的进度；实现 `#buildRecipes`
- `tags.TagsProvider` - 针对[标签]；实现 `#addTags`
- `advancements.AdvancementProvider` - 针对[进度]；向构造函数传递 `AdvancementSubProvider`

14.2 客户端资源（Assets）

14.2.1 模型生成

默认情况下，可以为模型或方块状态生成模型。每种都提供了一种生成必要JSON的方法（`ModelBuilder#toJson` 用于模型，`IGeneratedBlockState#toJson` 用于方块状态）。实现后，必须将关联的提供者 添加到 `DataGenerator` 中。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    DataGenerator gen = event.getGenerator();
    ExistingFileHelper efh = event.getExistingFileHelper();

    gen.addProvider(
        // 告诉生成器仅在生成客户端资源时运行
        event.includeClient(),
        output -> new MyItemModelProvider(output, MOD_ID, efh)
    );
    gen.addProvider(
        event.includeClient(),
        output -> new MyBlockStateProvider(output, MOD_ID, efh)
    );
}
```

模型文件

`ModelFile` 充当提供者引用或生成的所有模型的基础。每个模型文件存储相对于 `models` 子目录的位置，并可以断言该文件是否存在。

现存的模型文件

`ExistingModelFile` 是 `ModelFile` 的子类，它通过 `ExistingFileHelper#exists` 检查模型是否已存在于 `models` 子目录中。所有未生成的模型通常通过 `ExistingModelFile` 引用。

未检查的模型文件

`UncheckedModelFile` 是 `ModelFile` 的一个子类，它假定指定的模型存在于某个位置。

注意

不应存在使用 `UncheckedModelFile` 引用模型的情况。如果存在，则 `ExistingFileHelper` 无法正确跟踪关联的资源。

模型生成器

`ModelBuilder` 表示要生成的 `ModelFile`。它包含了关于模型的所有数据：它的父级、面、纹理、变换、照明和加载器。

提示

虽然可以生成复杂的模型，但建议事先使用建模软件构建这些模型。然后，数据提供者可以生成具有通过父复杂模型中定义的引用应用的特定纹理的子模型。

生成器的父级（通过 `ModelBuilder#parent`）可以是任何 `ModelFile`：生成的或现有的。一旦创建了生成器，生成的文件就会添加到 `ModelProvider` 中。生成器本身可以作为父级传入，也可以提供 `ResourceLocation`。

警告

如果在传递 ResourceLocation 时父模型不是在子模型之前生成的，则将引发异常。

模型中的每个元素（通过 ModelBuilder#element ）都被定义为使用两个三维点（分别为 ElementBuilder#from 和 #to ）的立方体，其中每个轴都被限制为值 [-16,32] （包括-16和32）。多维数据集的每个面（ ElementBuilder#face ）都可以指定面何时被剔除（ FaceBuilder#cullface ）、色调索引（ FaceBuilder#tintindex ）、来自 textures 键的纹理引用（ FaceBuilder#texture ）、纹理上的UV坐标（ FaceBuilder#uvs ）以及以90度间隔旋转（ FaceBuilder#rotation ）。

注意

建议在任何轴上元素超过 [0,16] 界限的方块模型分离为多个方块，例如多方块结构，以避免照明和剔除问题。

每个立方体还可以围绕指定点（ RotationBuilder#origin ）以22.5度的间隔（ RotationBuilder#angle ）为给定轴（ RotationBuilder#axis ）旋转（ ElementBuilder#rotation ）。立方体也可以相对于整个模型缩放所有面（ RotationBuilder#rescale ）。多维数据集还可以确定是否应渲染其阴影（ ElementBuilder#shade ）。

每个模型都定义了一个纹理键列表（ ModelBuilder#texture ），该列表指向一个位置或引用。然后，通过使用 # 前缀，可以在任何元素中引用每个键（ example 的纹理键可以在使用 #example 元素中引用）。位置指定纹理在 assets/<namespace>/textures/<path>.png 中的位置。引用由作为当前模型的子级的任何模型使用，作为以后定义纹理的键。

对于任何定义的透视图（在第一人称的左手、在图形用户界面、在地面等），还可以对模型进行转换（ ModelBuilder#transforms ）。对于任何透视图（ TransformsBuilder#transform ），都可以设置旋转（ TransformVecBuilder#rotation ）、平移（ TransformVecBuilder#translation ）和缩放（ TransformVecBuilder#scale ）。

最后，模型可以设置是否在某个存档（ ModelBuilder#ao ）中使用环境遮挡，以及从哪个位置从 ModelBuilder#guiLight 对模型进行明暗处理。

`BlockModelBuilder`

BlockModelBuilder 表示要生成的方块模型。除了 ModelBuilder 之外，还可以生成对整个模型的转换（ BlockModelBuilder#rootTransform ）。根可以围绕某个原点（ RootTransformBuilder#origin ）单独或全部在一个变换（ RootTransformBuilder#transform ）中进行平移（ RootTransformBuilder#transform ）、旋转（ RootTransformBuilder#rotation 、 RootTransformBuilder#postRotation ）和缩放（ RootTransformBuilder#origin ）。

`ItemModelBuilder`

ItemModelBuilder 表示要生成的物品模型。除了 ModelBuilder 之外，还可以生成overrides（ OverrideBuilder#override ）。应用于模型的每个重写都可以应用表示给定属性的条件，该属性必须高于指定值（ OverrideBuilder#predicate ）。如果满足条件，则将呈现指定的模型（ OverrideBuilder#model ），而不是此模型。

模型提供者

ModelProvider 子类负责生成构造的 ModelBuilder 。提供者接收生成器、mod id、要在其中生成的 models 文件夹中的子目录、 ModelBuilder 工厂和现有文件助手。每个提供器子类都必须实现 #registerModels 。

提供器包含创建 `ModelBuilder` 或为获取纹理或模型引用提供便利的基本方法：

方法	描述
`getBuilder`	Creates a new `ModelBuilder` within the provider's subdirectory for the given mod id.
`withExistingParent`	Creates a new `ModelBuilder` for the given parent. Should be used when the parent is not generated by the builder.
`mcLoc`	Creates a `ResourceLocation` for the path in the `minecraft` namespace.
`modLoc`	Creates a `ResourceLocation` for the path in the given mod id's namespace.

此外，还有几个助手可以使用普通模板轻松生成通用模型。大多数是方块模型，只有少数是通用的。

注意

尽管模型在一个特定的子目录中，但并不意味着该模型不能被另一个子目录中的模型引用。通常，它表示该模型用于该类型的对象。

`BlockModelProvider`

`BlockModelProvider` 用于通过 `block` 文件夹中的 `BlockModelBuilder` 生成方块模型。方块模型通常应为 `minecraft:block/block` 或其子模型之一的父模型，以便与物品模型一起使用。

注意

方块模型及其物品模型对应物通常不是通过 `BlockModelProvider` 和 `ItemModelProvider` 的直接子类生成的，而是通过 `BlockStateProvider` 生成的。

`ItemModelProvider`

`ItemModelProvider` 用于通过 `item` 文件夹中的 `ItemModelBuilder` 生成块模型。大多数物品模型的父级为 `item/generated`，并使用 `layer0` 来指定其纹理，这可以使用 `#singleTexture` 来完成。

注意

`item/generated` 可以支持堆叠在一起的五个纹理层：`layer0`、`layer1`、`layer2`、`layer3` 和 `layer4`。

```
// 在某个ItemModelProvider#registerModels中

// 将会生成'assets/<modid>/models/item/example_item.json'
// 父级将是'minecraft:item/generated'
// 对于纹理键'layer0'
//  其将会在'assets/<modid>/textures/item/example_item.png'
this.basicItem(EXAMPLE_ITEM.get());
```

注意

方块的物品模型通常应作为现有方块模型的父级，而不是为物品生成单独的模型。

方块状态提供者

`BlockStateProvider` 负责为所述方块生成 `blockstates` 中的方块状态JSON、 `models/block` 中的方块模型以及 `models/item` 中的物品模型。提供器接收数据生成器、mod id和现有的文件助手。每个 `BlockStateProvider` 子类都必须实现 `#registerStatesAndModels` 。

提供者包含用于生成方块状态JSON和方块模型的基本方法。物品模型必须单独生成，因为方块状态JSON可以定义多个模型以在不同的上下文中使用。然而，在处理更复杂的任务时，模组开发者应该注意一些常见的方法：

方法	描述
`models`	获取用于生成物品方块模型的 `BlockModelProvider` 。
`itemModels`	获取用于生成物品方块模型的 `ItemModelProvider` 。
`modLoc`	为给定mod id的命名空间中的路径创建 `ResourceLocation` 。
`mcLoc`	为 `minecraft` 命名空间中的路径创建 `ResourceLocation` 。
`blockTexture`	引用 `textures/block` 中与方块同名的纹理。
`simpleBlockItem`	为给定关联模型文件的方块创建物品模型。
`simpleBlockWithItem`	使用方块模型作为其父级，为方块模型和物品模型创建单个方块状态。

方块状态JSON由变量或条件组成。每个变量或条件都引用一个 `ConfiguredModelList` ： `ConfiguredModel` 的列表。每个配置的模型都包含模型文件（通过 `ConfiguredModel$Builder#modelFile` ）、90度间隔的X和Y旋转（分别通过 `#rotationX` 和 `rotationY` ）、当模型通过方块状态JSON旋转时纹理是否可以旋转（通过 `#uvLock` ），以及与列表中其他模型相比出现的模型的权重（通过 `#weight` ）。

生成器（ `ConfiguredModel#builder` ）还可以通过使用 `#nextModel` 创建下一个模型并重复设置直到调用 `#build` 来创建 `ConfiguredModel` 的数组。

`VariantBlockStateBuilder`

可以使用 `BlockStateProvider#getVariantBuilder` 生成变量。每个变体都指定了一个属性列表（ `PartialBlockstate` ），当该列表与存档中的 `BlockState` 匹配时，将显示从相应模型列表中选择的模型。如果存在未被定义的任何变体覆盖的 `BlockState` ，则抛出异常。对于任何 `BlockState` ，只有一种变体可以为true。

`PartialBlockstate` 通常使用以下三种方法之一进行定义：

方法	描述
`partialState`	创建要定义的 `PartialBlockstate` 。
`forAllStates`	定义一个函数，其中给定的 `BlockState` 可以由 `ConfiguredModel` 的数组表示。
`forAllStatesExcept`	定义一个类似于 `#forAllStates` 的函数；但是，它还指定了哪些属性不会影响渲染的模型。

对于 PartialBlockstate，可以指定定义的属性（#with）。 配置的模型可以设置（ #setModels），附加到现有模型（#addModels），或构建（#modelForState，然后是 ConfiguredModel$Builder#addModel，而不是 #ConfiguredModel$Builder#build）。

```
// 在某个BlockStateProvider#registerStatesAndModels中

// EXAMPLE_BLOCK_1：拥有属性BlockStateProperties#AXIS
this.getVariantBuilder(EXAMPLE_BLOCK_1) // 获取变量生成器
  .partialState() // 构建部分状态
  .with(AXIS, Axis.Y) // 当 BlockState AXIS = Y 时
    .modelForState() // 当 AXIS = Y 时设置模型
    .modelFile(yModelFile1) // 可以显示'yModelFile1'
    .nextModel() // 当 AXIS = Y 时添加另一个模型
    .modelFile(yModelFile2) // 可以显示'yModelFile2'
    .weight(2) // 此时将显示'yModelFile2' 2/3
    .addModel() // 完成当 AXIS = Y 时的模型
  .with(AXIS, Axis.Z) // 当 BlockState AXIS = Z 时
    .modelForState() // 当 AXIS = Z 时设置模型
    .modelFile(hModelFile) // 可以显示'hModelFile'
    .addModel() // 完成当 AXIS = Z 时的模型
  .with(AXIS, Axis.X)  // 当 BlockState AXIS = X 时
    .modelForState() // 当 AXIS = X 时设置模型
    .modelFile(hModelFile) // 可以显示'hModelFile'
    .rotationY(90) // 绕Y轴将'hModelFile'旋转90度
    .addModel(); // 完成当 AXIS = X 时的模型

// EXAMPLE_BLOCK_2：拥有属性BlockStateProperties#HORIZONTAL_FACING
this.getVariantBuilder(EXAMPLE_BLOCK_2) // 获取变量生成器
  .forAllStates(state -> // 对于全部可能的状态
    ConfiguredModel.builder() // 创建配置模型生成器
      .modelFile(modelFile) // 可以显示'modelFile'
      .rotationY((int) state.getValue(HORIZONTAL_FACING).toYRot()) // 根据变量需求将'modelFile'绕Y轴旋转
      .build() // 创建配置模型的数组
  );

// EXAMPLE_BLOCK_3：拥有属性BlockStateProperties#HORIZONTAL_FACING, BlockStateProperties#WATERLOGGED
this.getVariantBuilder(EXAMPLE_BLOCK_3) // 获取变量生成器
  .forAllStatesExcept(state -> // 对于全部HORIZONTAL_FACING状态
    ConfiguredModel.builder() // 创建配置模型生成器
      .modelFile(modelFile) // 可以显示'modelFile'
      .rotationY((int) state.getValue(HORIZONTAL_FACING).toYRot()) // 根据变量需求将'modelFile'绕Y轴旋转
      .build(), // 创建配置模型的数组
  WATERLOGGED); // 忽略WATERLOGGED属性
```

MultiPartBlockStateBuilder

可以使用 BlockStateProvider#getMultipartBuilder 生成多部分。每个部分（MultiPartBlockStateBuilder#part）指定一组属性条件，当与存档中的 BlockState 匹配时，将显示模型列表中的模型。所有与 BlockState 匹配的条件组将显示它们所选的模型。

对于任何部分（通过 ConfiguredModel$Builder#addModel 获得），当属性是指定值之一时，可以添加条件（通过 #condition）。条件必须全部成功，或者当设置了 #useOr 时，必须至少有一个成功。只要当前分组只包含其他组而不包含单个条件，就可以对条件进行分组（通过 #nestedGroup）。条件组可以使用 #endNestedGroup 留下，给定的部分可以通过 #end 完成。

```
// 在某个BlockStateProvider#registerStatesAndModels中
```

```
// 红石线
this.getMultipartBuilder(REDSTONE) // 获取多部分生成器
  .part() // 创建一个部分
    .modelFile(redstoneDot) // 可以显示'redstoneDot'
    .addModel() // 'redstoneDot'被显示，当...
    .useOr() // 这些条件中至少一个为true
    .nestedGroup() // 当所有组合条件均为true时，true
      .condition(WEST_REDSTONE, NONE) // 当WEST_REDSTONE为NONE时，true
      .condition(EAST_REDSTONE, NONE) // 当EAST_REDSTONE为NONE时，true
      .condition(SOUTH_REDSTONE, NONE) // 当SOUTH_REDSTONE为NONE时，true
      .condition(NORTH_REDSTONE, NONE) // 当NORTH_REDSTONE为NONE时，true
    .endNestedGroup() // 结束组合
    .nestedGroup() // 当所有组合条件均为true时，true
      .condition(EAST_REDSTONE, SIDE, UP) // 当EAST_REDSTONE为SIDE或UP时，true
      .condition(NORTH_REDSTONE, SIDE, UP) // 当NORTH_REDSTONE为SIDE或UP时，true
    .endNestedGroup() // 结束组合
    .nestedGroup() // 当所有组合条件均为true时，true
      .condition(EAST_REDSTONE, SIDE, UP) // 当EAST_REDSTONE为SIDE或UP时，true
      .condition(SOUTH_REDSTONE, SIDE, UP) // 当SOUTH_REDSTONE为SIDE或UP时，true
    .endNestedGroup() // 结束组合
    .nestedGroup() // 当所有组合条件均为true时，true
      .condition(WEST_REDSTONE, SIDE, UP) // 当WEST_REDSTONE为SIDE或UP时，true
      .condition(SOUTH_REDSTONE, SIDE, UP) // 当SOUTH_REDSTONE为SIDE或UP时，true
    .endNestedGroup() // 结束组合
    .nestedGroup() // 当所有组合条件均为true时，true
      .condition(WEST_REDSTONE, SIDE, UP) // 当WEST_REDSTONE为SIDE或UP时，true
      .condition(NORTH_REDSTONE, SIDE, UP) // 当NORTH_REDSTONE为SIDE或UP时，true
    .endNestedGroup() // 结束组合
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneSide0) // 可以显示'redstoneSide0'
    .addModel() // 'redstoneSide0'被显示，当...
    .condition(NORTH_REDSTONE, SIDE, UP) // NORTH_REDSTONE为SIDE或UP
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneSideAlt0) // 可以显示'redstoneSideAlt0'
    .addModel() // 'redstoneSideAlt0'被显示，当...
    .condition(SOUTH_REDSTONE, SIDE, UP) // SOUTH_REDSTONE为SIDE或UP
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneSideAlt1) // 可以显示'redstoneSideAlt1'
    .rotationY(270) // 将'redstoneSideAlt1'绕Y轴旋转270度
    .addModel() // 'redstoneSideAlt1'被显示，当...
    .condition(EAST_REDSTONE, SIDE, UP) // EAST_REDSTONE为SIDE或UP
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneSide1) // 可以显示'redstoneSide1'
    .rotationY(270) // 将'redstoneSide1'绕Y轴旋转270度
    .addModel() // 'redstoneSide1'被显示，当...
    .condition(WEST_REDSTONE, SIDE, UP) // WEST_REDSTONE为SIDE或UP
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneUp) // 可以显示'redstoneUp'
    .addModel() // 'redstoneUp'被显示，当...
    .condition(NORTH_REDSTONE, UP) // NORTH_REDSTONE为UP
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneUp) // 可以显示'redstoneUp'
    .rotationY(90) // 将'redstoneUp'绕Y轴旋转90度
```

```
    .addModel() // 'redstoneUp'被显示，当...
    .condition(EAST_REDSTONE, UP) // EAST_REDSTONE为UP
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneUp) // 可以显示'redstoneUp'
    .rotationY(180) // 将'redstoneUp'绕Y轴旋转180度
    .addModel() // 'redstoneUp'被显示，当...
    .condition(SOUTH_REDSTONE, UP) // SOUTH_REDSTONE为UP
    .end() // 结束该部分
  .part() // 创建一个部分
    .modelFile(redstoneUp) // 可以显示'redstoneUp'
    .rotationY(270) // 将'redstoneUp'绕Y轴旋转270度
    .addModel() // 'redstoneUp'被显示，当...
    .condition(WEST_REDSTONE, UP) // WEST_REDSTONE为UP
    .end(); // 结束该部分
```

模型加载器生成器

还可以为给定的 `ModelBuilder` 生成自定义模型加载器。自定义模型加载器子类 `CustomLoaderBuilder` ，可以通过 `#customLoader` 应用于 `ModelBuilder` 。传入的工厂方法创建了一个新的加载器生成器，可以对其进行配置。完成所有更改后，自定义加载器可以通过 `CustomLoaderBuilder#end` 返回到 `ModelBuilder` 。

模型生成器	工厂方法	描述
`DynamicFluidContainerModelBuilder`	`#begin`	为特定的流体生成一个桶模型。
`CompositeModelBuilder`	`#begin`	生成一个由模型组成的模型。
`ItemLayersModelBuilder`	`#begin`	生成一个 `item/generated` 模型的Forge实现。
`SeparateTransformsModelBuilder`	`#begin`	生成一个模型，其修改基于特定的变换。
`ObjModelBuilder`	`#begin`	生成一个OBJ模型。

```
// 对于某个BlockModelBuilder生成器
builder.customLoader(ObjModelBuilder::begin) // 自定义加载器'forge:obj'
  .modelLocation(modLoc("models/block/model.obj")) // 设置OBJ模型位置
  .flipV(true) // 在提供的.mtl纹理中翻转V坐标
  .end() // 完成自定义加载器配置
.texture("particle", mcLoc("block/dirt")) // 将粒子纹理设置为泥土
.texture("texture0", mcLoc("block/dirt")); // 将'texture0'纹理设置为泥土
```

自定义模型加载器生成器

可以通过扩展 `CustomLoaderBuilder` 来创建自定义加载器生成器。构造函数仍然可以具有 `protected` 的可见性，其中 `ResourceLocation` 硬编码为通过 `ModelEvent$RegisterGeometryLoaders#register` 注册的加载器id。然后，可以通过静态工厂方法或构造函数（如果设置为 `public` ）初始化生成器。

```
public class ExampleLoaderBuilder<T extends ModelBuilder<T>> extends CustomLoaderBuilder<T> {
  public static <T extends ModelBuilder<T>> ExampleLoaderBuilder<T> begin(T parent, ExistingFileHelper existingFileHelper) {
    return new ExampleLoaderBuilder<>(parent, existingFileHelper);
  }

  protected ExampleLoaderBuilder(T parent, ExistingFileHelper existingFileHelper) {
    super(new ResourceLocation(MOD_ID, "example_loader"), parent, existingFileHelper);
```

```
  }
}
```

Afterwards, any configurations specified by the loader should be added as chainable methods.

```
// 在ExampleLoaderBuilder中
public ExampleLoaderBuilder<T> exampleInt(int example) {
  // 设置int
  return this;
}

public ExampleLoaderBuilder<T> exampleString(String example) {
  // 设置string
  return this;
}
```

If any additional configuration is specified, `#toJson` should be overridden to write the additional properties.

```
// 在ExampleLoaderBuilder中
@Override
public JsonObject toJson(JsonObject json) {
  json = super.toJson(json); // 处理基础加载器属性
  // 编码自定义加载器属性
  return json;
}
```

自定义模型提供者

自定义模型提供者需要 `ModelBuilder` 子类和 `ModelProvider` 子类，前者定义要生成的模型的基础，后者生成模型。

`ModelBuilder` 子类包含任何特殊属性，这些属性可以专门应用于这些类型的模型（物品模型可以具有重写）。如果添加了任何附加属性，则需要重写 `#toJson` 以写入附加信息。

```
public class ExampleModelBuilder extends ModelBuilder<ExampleModelBuilder> {
  // ...
}
```

`ModelProvider` 子类不需要特殊的逻辑。构造函数应硬编码 `models` 文件夹和 `ModelBuilder` 中的子目录，以表示要生成的模型。

```
public class ExampleModelProvider extends ModelProvider<ExampleModelBuilder> {

  public ExampleModelProvider(PackOutput output, String modid, ExistingFileHelper existingFileHelper) {
    // 如果'#getBuilder'中未指定'modid'，则模型将生成到'assets/<modid>/models/example'
    super(output, modid, "example", ExampleModelBuilder::new, existingFileHelper);
  }
}
```

自定义模型**Consumer**

自定义模型Consumer，如 `BlockStateProvider` ，可以通过手动生成模型来创建。应指定用于生成模型的 `ModelProvider` 子类并使其可用。

```
public class ExampleModelConsumerProvider implements IDataProvider {
```

```
  public ExampleModelConsumerProvider(PackOutput output, String modid, ExistingFileHelper existingFileHelper) {
    this.example = new ExampleModelProvider(output, modid, existingFileHelper);
  }
}
```

一旦数据提供者运行，就可以使用 `ModelProvider#generateAll` 生成 `ModelProvider` 子类中的模型。

```
// 在ExampleModelConsumerProvider中
@Override
public CompletableFuture<?> run(CachedOutput cache) {
  // 填入模型提供者
  CompletableFuture<?> exampleFutures = this.example.generateAll(cache); // 生成模型

  // 运行逻辑并创建CompletableFuture以写入文件
  // ...

  // 假设我们有一个新的CompletableFuture providerFuture
  return CompletableFuture.allOf(exampleFutures, providerFuture);
}
```

14.2.2 语言生成

可以通过子类化 `LanguageProvider` 并实现 `#addTranslations` 为模组生成语言文件。创建的每个 `LanguageProvider` 子类代表一个单独的locale（ `en_us` 代表美式英语， `es_es` 代表西班牙语等）。实现后，必须将提供者添加到 `DataGenerator` 中。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成客户端资源时运行
        event.includeClient(),
        // 美式英语的本地化
        output -> new MyLanguageProvider(output, MOD_ID, "en_us")
    );
}
```

LanguageProvider

每个语言提供者都是一个简单的字符串映射，其中每个翻译键都映射到一个本地化名称。可以使用 `#add` 添加翻译键映射。此外，还有一些方法使用 `Block` 、 `Item` 、 `ItemStack` 、 `Enchantment` 、 `MobEffect` 和 `EntityType` 的翻译键。

```
// 在LanguageProvider#addTranslations中
this.addBlock(EXAMPLE_BLOCK, "Example Block");
this.add("object.examplemod.example_object", "Example Object");
```

提示

包含非美式英语字母数字值的本地化名称可以按原样提供。提供者会自动将字符翻译为等效的unicode，供游戏读取。

```
// 编码为'Example with a d\u00EDacritic'
this.addItem("example.diacritic", "Example with a díacritic");
```

14.2.3 音效定义生成

通过子类化 `SoundDefinitionsProvider` 并实现 `#registerSounds`，可以为模组生成 `sounds.json` 文件。实现后，必须将提供者添加到 `DataGenerator` 中。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成客户端资源时运行
        event.includeClient(),
        output -> new MySoundDefinitionsProvider(output, MOD_ID, event.getExistingFileHelper())
    );
}
```

添加一个音效

可以通过 `#add` 指定音效名称和定义来生成音效定义。音效名称可以从 `SoundEvent` 、 `ResourceLocation` 或字符串中提供。

> **警告**
>
> 提供的音效名称将始终假定命名空间是提供给提供者的构造函数的mod id。没有对音效名称的命名空间执行验证！

`SoundDefinition`

可以使用 `#definition` 创建 `SoundDefinition` 。定义包含用于定义音效实例的数据。

定义指定了一些方法：

方法	描述
`with`	添加选择定义时可能播放的音效。
`subtitle`	设置定义的翻译键。
`replace`	当为 `true` 时，将删除其他 `sounds.json` 为该定义定义的音效，而不是附加到该定义。

`SoundDefinition$Sound`

提供给 `SoundDefinition` 的音效可以使用 `SoundDefinitionsProvider#sound` 指定。这些方法采用音效的引用和 `SoundType` （如果已指定）。

`SoundType` 可以是两个值之一：

音效类型	定义
`SOUND`	指定位于 `assets/<namespace>/sounds/<path>.ogg` 的音效的一个引用。
`EVENT`	指定由 `sounds.json` 定义的另一个音效的名称的引用。

从 `SoundDefinitionsProvider#sound` 创建的每个 `Sound` 都可以指定关于如何加载和播放所提供音效的其他配置：

方法	描述
`volume`	设置音效的音量大小，必须大于 `0` 。
`pitch`	设置音效的音高大小，必须大于 `0` 。
`weight`	设置音效被选定时播放音效的可能性。
`stream`	当为 `true` 时，从文件中读取音效，而不是将音效加载到内存中。推荐用于长音效：背景音乐、音乐唱片等。
`attenuationDistance`	设置可以听到音效的所距离的方块数。
`preload`	当为 `true` 时，一旦加载资源包，就会立即将音效加载到内存中。

```
// 在某个SoundDefinitionsProvider#registerSounds中
this.add(EXAMPLE_SOUND_EVENT, definition()
  .subtitle("sound.examplemod.example_sound") // 设置翻译键
  .with(
    sound(new ResourceLocation(MODID, "example_sound_1")) // 设置第一个音效
      .weight(4) // 具有4 / 5 = 80%的播放机率
      .volume(0.5), // 将调用此音效的所有音量缩放一半
    sound(new ResourceLocation(MODID, "example_sound_2")) // 设置第二个音效
      .stream() // 流播该音效
  )
);

this.add(EXAMPLE_SOUND_EVENT_2, definition()
  .subtitle("sound.examplemod.example_sound") // 设置翻译键
  .with(
    sound(EXAMPLE_SOUND_EVENT.getLocation(), SoundType.EVENT) // 从'EXAMPLE_SOUND_EVENT'添加音效
      .pitch(0.5) // 将调用此音效的所有音高缩放一半
  )
);
```

14.3 服务端数据（Data）

14.3.1 配方生成

可以通过子类化 `RecipeProvider` 并实现 `#buildRecipes` 来为模组生成配方。一旦Consumer接受 `FinishedRecipe` 视图，就会提供一个用于生成数据的配方。 `FinishedRecipe` 既可以手动创建和提供，也可以为方便起见，使用 `RecipeBuilder` 创建。

实现后，该提供者必须被添加到 `DataGenerator` 。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成服务端资源时运行
        event.includeServer(),
        MyRecipeProvider::new
    );
}
```

RecipeBuilder

`RecipeBuilder` 是一个方便的实现，用于创建要生成的 `FinishedRecipe` 。它提供了解锁、分组、保存和获取配方结果的基本定义。这分别通过 `#unlockedBy` 、 `#group` 、 `#save` 和 `#getResult` 来完成。

> **重要**
>
> 原版配方生成器中不支持配方中的 `ItemStack` 输出。对于现有的原版配方序列化器，必须以不同的方式构建 `FinishedRecipe` 才能生成此数据。

> **警告**
>
> 正在生成的物品结果必须指定有效的 `RecipeCategory` ；否则，将引发 `NullPointerException` 。

除[`SpecialRecipeBuilder`]外的所有配方构建器都需要指定一个进度标准。如果玩家以前使用过配方，则所有配方都会生成解锁配方的标准。然而，必须指定一个额外的标准，允许玩家在没有任何先验知识的情况下获得配方。如果指定的任何标准为真，则玩家将获得配方书的配方。

> **提示**
>
> 配方标准通常使用 `InventoryChangeTrigger` 在用户物品栏中存在某些物品时解锁配方。

ShapedRecipeBuilder

`ShapedRecipeBuilder` 用于生成有序配方。该生成器可以通过 `#shaped` 进行初始化。保存前可以指定配方组、输入符号模式、配料的符号定义和配方解锁条件。

```
// 在RecipeProvider#buildRecipes(writer)中
ShapedRecipeBuilder builder = ShapedRecipeBuilder.shaped(RecipeCategory.MISC, result)
  .pattern("a a") // 创建配方图案
  .define('a', item) // 定义符号代表什么
```

```
  .unlockedBy("criteria", criteria) // 该配方如何解锁
  .save(writer); // 将数据加入生成器
```

附加验证检查

有序配方在构建前进行了一些额外的验证检查：

- 图案必须被定义且接受多于一个物品。
- 所有图案行的宽度必须相同。
- 一个符号不能被定义多次。
- 空格字符（`' '`）被保留用于表示格中无物品，因此无法被定义。
- 图案必须使用用户定义的全部符号。

ShapelessRecipeBuilder

`ShapelessRecipeBuilder` 用于生成无序配方。该生成器可以通过 `#shapeless` 进行初始化。保存前可以指定配方组、输入原料和配方解锁条件。

```
// 在RecipeProvider#buildRecipes(writer)中
ShapelessRecipeBuilder builder = ShapelessRecipeBuilder.shapeless(RecipeCategory.MISC, result)
  .requires(item) // 将物品加入配方
  .unlockedBy("criteria", criteria) // 该配方如何解锁
  .save(writer); // 将数据加入生成器
```

SimpleCookingRecipeBuilder

`SimpleCookingRecipeBuilder` 用于生成熔炼、高炉熔炼、烟熏和篝火烹饪配方。此外，使用 `SimpleCookingSerializer` 的自定义烹饪配方也可以是使用该生成器生成的数据。生成器可以分别通过 `#smelting` 、 `#blasting` 、 `#smoking` 、 `#campfireCooking` 或 `#cooking` 进行初始化。保存前可以指定配方组和配方解锁条件。

```
// 在RecipeProvider#buildRecipes(writer)中
SimpleCookingRecipeBuilder builder = SimpleCookingRecipeBuilder.smelting(input, RecipeCategory.MISC, result, experience, cook
  .unlockedBy("criteria", criteria) // 该配方如何解锁
  .save(writer); // 将数据加入生成器
```

SingleItemRecipeBuilder

`SingleItemRecipeBuilder` 用于生成切石配方。此外，使用类似 `SingleItemRecipe$Serializer` 的序列化器的自定义但物品配方也可以是使用该生成器生成的数据。生成器可以分别通过 `#stonecutting` 或通过构造函数进行初始化。保存前可以指定配方组和配方解锁条件。

```
// 在RecipeProvider#buildRecipes(writer)中
SingleItemRecipeBuilder builder = SingleItemRecipeBuilder.stonecutting(input, RecipeCategory.MISC, result)
  .unlockedBy("criteria", criteria) // 该配方如何解锁
  .save(writer); // 将数据加入生成器
```

非 `RecipeBuilder` 生成器

由于缺少前面提到的所有配方所使用的功能，一些配方生成器没有实现 `RecipeBuilder` 。

SmithingTransformRecipeBuilder

`SmithingTransformRecipeBuilder` 用于生成转换物品的锻造配方。此外，使用序列化器（如 `SmithingTransformRecipe$Serializer`）的自定义配方也可以是使用此生成器生成的数据。生成器可以分别通过 `#smithing` 或通过构造函数进行初始化。保存前可以指定配方解锁条件。

```
// 在RecipeProvider#buildRecipes(writer)中
SmithingTransformRecipeBuilder builder = SmithingTransformRecipeBuilder.smithing(template, base, addition, RecipeCategory.MIS(
  .unlocks("criteria", criteria) // 该配方如何解锁
  .save(writer, name); // 将数据加入生成器
```

SmithingTrimRecipeBuilder

`SmithingTrimRecipeBuilder` 用于生成盔甲装饰的锻造配方。此外，使用类似 `SmithingTrimRecipe$Serializer` 的序列化器的自定义升级配方也可以是使用该生成器生成的数据。生成器可以分别通过 `#smithingTrim` 或通过构造函数进行初始化。保存前可以指定配方解锁条件。

```
// 在RecipeProvider#buildRecipes(writer)中
SmithingTrimRecipe builder = SmithingTrimRecipe.smithingTrim(template, base, addition, RecipeCategory.MISC)
  .unlocks("criteria", criteria) // 该配方如何解锁
  .save(writer, name); // 将数据加入生成器
```

SpecialRecipeBuilder

`SpecialRecipeBuilder` is used to generate empty JSONs for dynamic recipes that cannot easily be constrained to the recipe JSON format (dying armor, firework, etc.). The builder can be initialized via `#special`.

```
// 在RecipeProvider#buildRecipes(writer)中
SpecialRecipeBuilder.special(dynamicRecipeSerializer)
  .save(writer, name); // 将数据加入生成器
```

条件性配方

条件性配方也可以是通过 `ConditionalRecipe$Builder` 生成的数据。生成器可以使用 `#builder` 获得。

每个配方的条件可以通过首先调用 `#addCondition`，然后在指定所有条件后调用 `#addRecipe` 来指定。这个过程可以重复多次，只要程序员愿意。

在指定了所有配方后，可以在最后使用 `#generateAdvancement` 为每个配方添加进度。或者，可以使用 `#setAdvancement` 设置条件性进度。

```
// 在RecipeProvider#buildRecipes(writer)中
ConditionalRecipe.builder()
  // 为该配方添加条件
  .addCondition(...)
  // 添加当条件为true时返回的配方
  .addRecipe(...)

  // 为下一个配方添加接下来的条件
  .addCondition(...)
  // 添加当条件为true时返回的下一个配方
  .addRecipe(...)

  // 创建条件性进度，其使用上面配方中的条件和所解锁的进度
  .generateAdvancement()
  .build(writer, name);
```

IConditionBuilder

为了简化向条件配方添加条件而不必手动构造每个条件实例，扩展的 `RecipeProvider` 可以实现 `IConditionBuilder` 。该接口添加了可以轻松构造条件实例的方法。

```
// 在ConditionalRecipe$Builder#addCondition中
(
  // 如果'examplemod:example_item'
  // 或（OR）'examplemod:example_item2'存在
  // 并且（AND）
  // 非FALSE（NOT FALSE）

  // Methods are defined by IConditionBuilder
  and(
    or(
      itemExists("examplemod", "example_item"),
      itemExists("examplemod", "example_item2")
    ),
    not(
      FALSE()
    )
  )
)
```

自定义配方序列化器

自定义配方序列化器可以是通过创建可以构造 `FinishedRecipe` 的生成器生成的数据。完成的配方将配方数据及其所解锁的进度（如果存在）编码为JSON。此外，还指定了配方的名称和序列化器，以了解在加载时向何处写入以及可以解码对象的内容。构造完 `FinishedRecipe` 后，只需将其传递给 `RecipeProvider#buildRecipes` 提供的 `Consumer` 。

提示

`FinishedRecipe` 足够灵活，任何对象转换都可以是数据生成的，而不仅仅是物品。

14.3.2 战利品表生成

可以通过构造新的 `LootTableProvider` 并提供 `LootTableProvider$SubProviderEntry` 来为模组生成战利品表。该提供者必须被添加到 `DataGenerator` 中。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成服务端资源时运行
        event.includeServer(),
        output -> new MyLootTableProvider(
          output,
          // 指定需要生成的表的注册表名称，或者可留空
          Collections.emptySet(),
          // 生成战利品的子提供者
          List.of(subProvider1, subProvider2, /*...*/)
        )
    );
}
```

LootTableSubProvider

每个 `LootTableProvider$SubProviderEntry` 接受一个提供的 `LootTableSubProvider`，该 `LootTableSubProvider` 为给定的 `LootContextParamSet` 生成战利品表。`LootTableSubProvider` 包含一个方法，该方法采用编写器（`BiConsumer<ResourceLocation, LootTable.Builder>`）来生成表。

```
public class ExampleSubProvider implements LootTableSubProvider {

  // 用于为包装Supplier创建工厂方法
  public ExampleSubProvider() {}

  // 用于生成战利品表的方法
  @Override
  public void generate(BiConsumer<ResourceLocation, LootTable.Builder> writer) {
    // 在此处通过调用writer#accept生成战利品表
  }
}
```

The table can then be added to `LootTableProvider#getTables` for any available `LootContextParamSet`:

```
// 在将会传递给LootTableProvider构造函数的列表中
new LootTableProvider.SubProviderEntry(
  ExampleSubProvider::new,
  // 'empty'参数集的战利品表生成器
  LootContextParamSets.EMPTY
)
```

BlockLootSubProvider 和 EntityLootSubProvider 子类

对于 `LootContextParamSets#BLOCK` 和 `#ENTITY`，有一些特殊类型（分别为 `BlockLootSubProvider` 和 `EntityLootSubProvider`），它们提供了额外的帮助方法来创建和验证是否存在战利品表。

`BlockLootSubProvider` 的构造函数接受一个物品列表和一个 `FeatureFlagSet`，前者是耐爆炸的，用于确定如果方块爆炸，是否可以生成战利品表，后者用于确定是否启用了该方块，以便为其生成战利品表。

```
// 在某个BlockLootSubProvider子类中
public MyBlockLootSubProvider() {
  super(Collections.emptySet(), FeatureFlags.REGISTRY.allFlags());
}
```

`EntityLootSubProvider` 的构造函数接受一个 `FeatureFlagSet`，它确定是否启用了实体类型，以便为其生成战利品表。

```
// 在某个EntityLootSubProvider子类中
public MyEntityLootSubProvider() {
  super(FeatureFlags.REGISTRY.allFlags());
}
```

要使用它们，所有注册的对象必须分别提供给 `BlockLootSubProvider#getKnownBlocks` 和 `EntityLootSubProvider#getKnownEntityTypes`。这些方法是为了确保Iterable中的所有对象都有一个战利品表。

提示

如果 `DeferredRegister` 用于注册模组的对象，则可以通过 `DeferredRegister#getEntries` 向 `#getKnown*` 方法提供条目：

```
// 在针对某个DeferredRegister BLOCK_REGISTRAR的某个BlockLootSubProvider子类中
@Override
protected Iterable<Block> getKnownBlocks() {
  return BLOCK_REGISTRAR.getEntries() // 获取所有已注册的条目
    .stream() // 流播所有已包装的对象
    .flatMap(RegistryObject::stream) // 如果可行，获取该对象
    ::iterator; // 创建该Iterable
}
```

战利品表本身可以通过实现 `#generate` 方法来添加。

```
// 在某个BlockLootSubProvider子类中
@Override
public void generate() {
  // 在此处添加战利品表
}
```

战利品表生成器

要生成战利品表，它们被 `LootTableSubProvider` 接受为 `LootTable$Builder`。之后，在 `LootTableProvider$SubProviderEntry` 中设置指定的 `LootContextParamSet`，然后通过 `#build` 生成。在构建之前，生成器可以指定影响战利品表功能的条目、条件和修改器。

注意

战利品表的功能非常广泛，因此本文档不会对其进行全面介绍。取而代之的是，将对每个组件进行简要描述。每个组件的特定子类型可以使用IDE找到。它们的实现将留给读者练习。

LootTable

战利品表是基本对象，可以使用 `LootTable#lootTable` 将其转换为所需的 `LootTable$Builder`。战利品表可以通过池列表（通过 `#withPool`）以及修改这些池的结果物品的功能（通过 `#apply`）来构建，池列表按指定的顺序应用。

LootPool

战利品池代表一个执行操作的组，并且可以使用 `LootPool#lootPool` 生成 `LootPool$Builder` 。每个战利品池都可以指定定义池中操作的条目（通过 `#add` ）、定义是否应该执行池中的操作的条件（通过 `#when` ）以及修改条目的结果物品的功能（通过 `#apply` ）。每个池可以按指定次数执行（通过 `#setRolls` ）。此外，还可以指定奖金执行（通过 `#setBonusRolls` ），这取决于执行者的运气。

LootPoolEntryContainer

战利品条目定义了选择时要执行的操作，通常是生成物品。每个条目都有一个关联的已注册的 `LootPoolEntryType` 。它们也有自己的关联生成器，为 `LootPoolEntryContainer$Builder` 的子类型。多个条目可以同时执行（通过 `#append` ）或顺序执行，直到一个条目失败为止（通过 `#then` ）。此外，条目可以在失败时默认为另一个条目（通过 `#otherwise` ）。

LootItemCondition

战利品条件定义了执行某些操作所需满足的要求。每个条件都有一个关联的已注册的 `LootItemConditionType` 。它们也有自己的关联生成器，为 `LootItemCondition$Builder` 的子类型。默认情况下，所有指定的战利品条件都必须返回true才能执行操作。战利品条件也可以指定为只有一个必须返回true（通过 `#or` ）。此外，条件的结果输出可以反转（通过 `#invert` ）。

LootItemFunction

战利品函数在将执行结果传递给输出之前会对其进行修改。每个函数都有一个关联的已注册的 `LootItemFunctionType` 。它们也有自己的关联生成器，为 `LootItemFunction$Builder` 的子类型。

NbtProvider

NBT提供者是由 `CopyNbtFunction` 定义的一种特殊类型的函数。它们定义了从何处提取标记信息。每个提供者都有一个关联的已注册的 `LootNbtProviderType` 。

NumberProvider

数字提供者决定战利品池执行的次数。每个提供者都有一个关联的已注册的 `LootNumberProviderType` 。

ScoreboardNameProvider

记分牌提供者是由 `ScoreboardValue` 定义的一种特殊类型的数字提供者。他们定义了记分牌的名称，以获取要执行的掷数。每个提供者都有一个关联的已注册的 `LootScoreProviderType` 。

14.3.3 标签生成

可以通过子类化 `TagsProvider` 并实现 `#addTags` 来为模组生成标签。实现后，该提供者必须被添加到 `DataGenerator` 中。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成服务端资源时运行
        event.includeServer(),
        // 扩展net.minecraftforge.common.data.BlockTagsProvider
        output -> new MyBlockTagsProvider(
          output,
          event.getLookupProvider(),
          MOD_ID,
          event.getExistingFileHelper()
        )
    );
}
```

TagsProvider

标签提供者有两种用于生成标签的方法：通过 `#tag` 创建带有对象和其他标签的标签，或通过 `#getOrCreateRawBuilder` 使用其他对象类型的标签生成标签数据。

> **注意**
>
> 通常，提供者不会直接调用 `#getOrCreateRawBuilder`，除非注册表包含来自不同注册表的对象表示（方块具有物品表示以获得物品栏中的方块）。

当调用 `#tag` 时，将创建一个 `TagAppender`，它充当要添加到标签中的元素的可链接Consumer：

方法	描述
`add`	通过对象的资源键将对象添加到标签中。
`addOptional`	通过对象的名称将对象添加到标签中。如果对象不存在，则加载时将跳过该对象。
`addTag`	通过标签键将标签添加到标签中。内部标签中的所有元素现在都是外部标签的一部分。
`addOptionalTag`	通过标签的名称将标签添加到标签中。如果标签不存在，则加载时将跳过该标签。
`replace`	当为 `true` 时，从其他数据包添加到此标签的所有先前加载的条目都将被丢弃。如果在这个数据包之后加载了一个数据包，那么它仍然会将条目附加到标签中。
`remove`	通过对象或标签的名称或键从标签中删除对象或标签。

```
// 在某个TagProvider#addTags中
this.tag(EXAMPLE_TAG)
  .add(EXAMPLE_OBJECT) // 向该标签添加一个对象
  .addOptional(new ResourceLocation("othermod", "other_object")) // 向该标签添加一个来自其他模组的对象

this.tag(EXAMPLE_TAG_2)
```

```
.addTag(EXAMPLE_TAG) // 向该标签添加一个标签
.remove(EXAMPLE_OBJECT) // 从该标签中移除一个对象
```

重要

如果模组的标签软依赖于另一个模组的标签（另一个模组可能在运行时存在，也可能不存在），则应使用可选方法引用其他模组的标签。

Existing Providers

Minecraft包含一些用于某些注册表的标签提供者，这些注册表可以被子类化。此外，一些提供者包含额外的辅助方法，以便更容易地创建标签。

注册表对象类型	标签提供者
`Block`	`BlockTagsProvider` *
`Item`	`ItemTagsProvider`
`EntityType`	`EntityTypeTagsProvider`
`Fluid`	`FluidTagsProvider`
`GameEvent`	`GameEventTagsProvider`
`Biome`	`BiomeTagsProvider`
`FlatLevelGeneratorPreset`	`FlatLevelGeneratorPresetTagsProvider`
`WorldPreset`	`WorldPresetTagsProvider`
`Structure`	`StructureTagsProvider`
`PoiType`	`PoiTypeTagsProvider`
`BannerPattern`	`BannerPatternTagsProvider`
`CatVariant`	`CatVariantTagsProvider`
`PaintingVariant`	`PaintingVariantTagsProvider`
`Instrument`	`InstrumentTagsProvider`
`DamageType`	`DamageTypeTagsProvider`

* `BlockTagsProvider` 是一个由Forge添加的 `TagsProvider` 。

`ItemTagsProvider#copy`

方块具有用于在物品栏中获取它们的物品表示。因此，许多方块标签也可以是物品标签。为了容易地生成与方块标签具有相同条目的物品标签，可以使用 `#copy` 方法，该方法接受要从中复制的方块标签和要复制到的物品标签。

```
// 在ItemTagsProvider#addTags中
this.copy(EXAMPLE_BLOCK_TAG, EXAMPLE_ITEM_TAG);
```

自定义标签提供者

可以通过 `TagsProvider` 子类创建自定义标签提供者，该子类接受注册表键来为其生成标签。

```
public RecipeTypeTagsProvider(PackOutput output, CompletableFuture<HolderLookup.Provider> registries, ExistingFileHelper fileH
  super(output, Registries.RECIPE_TYPE, registries, MOD_ID, fileHelper);
}
```

Intrinsic Holder Tags Providers

一种特殊类型的 `TagProvider` 是 `IntrinsicHolderTagsProvider` 。当通过 `#tag` 使用此提供者创建标签时，可以使用对象本身通过 `#add` 将自己添加到标签中。为此，在构造函数中提供了一个函数，将对象转换为其 `ResourceKey` 。

```
// `IntrinsicHolderTagsProvider`的子类型
public AttributeTagsProvider(PackOutput output, CompletableFuture<HolderLookup.Provider> registries, ExistingFileHelper fileHe
  super(
    output,
    ForgeRegistries.Keys.ATTRIBUTES,
    registries,
    attribute -> ForgeRegistries.ATTRIBUTES.getResourceKey(attribute).get(),
    MOD_ID,
    fileHelper
  );
}
```

14.3.4 进度生成

可以通过构建新的 `AdvancementProvider` 并提供 `AdvancementSubProvider` 来为模组生成进度。进度既可以手动创建和提供，也可以为方便起见，使用 `Advancement$Builder` 创建。该提供者必须被添加到 `DataGenerator` 中。

注意

Forge为 `AdvancementProvider` 提供了一个名为 `ForgeAdvancementProvider` 的扩展，它可以更好地集成以生成进度。因此，本文档将使用 `ForgeAdvancementProvider` 和子提供者接口 `ForgeAdvancementProvider$AdvancementGenerator` 。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成服务端资源时运行
        event.includeServer(),
        output -> new ForgeAdvancementProvider(
          output,
          event.getLookupProvider(),
          event.getExistingFileHelper(),
          // 生成进度的子提供者
          List.of(subProvider1, subProvider2, /*...*/)
        )
    );
}
```

ForgeAdvancementProvider$AdvancementGenerator

`ForgeAdvancementProvider$AdvancementGenerator` 负责生成进度，包含一个接受注册表查找的方法、写入器（ `Consumer<Advancement>` ）和现有文件助手..

```
// 在ForgeAdvancementProvider$AdvancementGenerator的某个子类中，或作为一个lambda引用

@Override
public void generate(HolderLookup.Provider registries, Consumer<Advancement> writer, ExistingFileHelper existingFileHelper) {
  // 在此处构建进度
}
```

Advancement$Builder

`Advancement$Builder` 是一个方便的实现，用于创建要生成的 `Advancement` 。它允许定义父级进度、显示信息、进度完成时的奖励以及解锁进度的要求。只需指定要求即可创建 `Advancement` 。

尽管不是必需的，但有许多方法很重要：

方法	描述
`parent`	设置此进度直接链接到的进度。可以指定进度的名称，也可以指定进度本身（如果它是由模组开发者生成的）。
`display`	设置要显示在聊天、toast和进度屏幕上的信息。
`rewards`	设置此进度完成时获得的奖励。
`addCriterion`	为此进度添加一个条件。
`requirements`	指定是所有条件都必须返回true，还是至少有一个条件必须返回true。可以使用额外的重载来混合和匹配这些操作。

一旦准备好构建 `Advancement$Builder`，就应该调用 `#save` 方法，该方法接受写入器、进度的注册表名以及用于检查提供的父级是否存在的文件助手。

```
// 在某个ForgeAdvancementProvider$AdvancementGenerator#generate(registries, writer, existingFileHelper)中
Advancement example = Advancement.Builder.advancement()
  .addCriterion("example_criterion", triggerInstance) // 该进度如何解锁
  .save(writer, name, existingFileHelper); // 将数据加入生成器
```

14.3.5 全局战利品修改器生成

可以通过子类化 `GlobalLootModifierProvider` 并实现 `#start` 来为模组生成全局战利品修改器（GLM）。每个GLM都可以通过调用 `#add` 并指定要序列化的修改器和修改器实例的名称来添加生成。实现后，该提供者必须被添加到 `DataGenerator` 中。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成服务端资源时运行
        event.includeServer(),
        output -> new MyGlobalLootModifierProvider(output, MOD_ID)
    );
}

// 在某个GlobalLootModifierProvider#start中
this.add("example_modifier", new ExampleModifier(
  new LootItemCondition[] {
    WeatherCheck.weather().setRaining(true).build() // 当下雨时执行
  },
  "val1",
  10,
  Items.DIRT
));
```

14.3.6 数据包注册表对象生成

Datapack registry objects can be generated for a mod by constructing a new `DatapackBuiltinEntriesProvider` and providing a `RegistrySetBuilder` with the new objects to register. The provider must be added to the `DataGenerator`. 通过构造新的 `DatapackBuiltinEntriesProvider` 并为 `RegistrySetBuilder` 提供要注册的新对象，可以为模组生成数据包注册表对象。该提供者必须被添加到 `DataGenerator` 中。

注意

`DatapackBuiltinEntriesProvider` 是 `RegistriesDatapackGenerator` 之上的一个Forge扩展，它可以正确处理引用现有数据包注册表对象而不会分解条目。因此，本文档将使用 `DatapackBuiltinEntriesProvider`。

```
// 在模组事件总线上
@SubscribeEvent
public void gatherData(GatherDataEvent event) {
    event.getGenerator().addProvider(
        // 告诉生成器仅在生成服务端资源时运行
        event.includeServer(),
        output -> new DatapackBuiltinEntriesProvider(
          output,
          event.getLookupProvider(),
          // 包含要生成的数据包注册表对象的生成器
          new RegistrySetBuilder().add(/* ... */),
          // 用于生成的数据包注册表对象的mod id集合
          Set.of(MOD_ID)
        )
    );
}
```

RegistrySetBuilder

`RegistrySetBuilder` 负责构建游戏中使用的所有数据包注册表对象。生成器可以为注册表添加一个新条目，然后注册表可以将对象注册到该注册表中。

首先，可以通过调用构造函数来初始化 `RegistrySetBuilder` 的新实例。然后，可以调用 `#add` 方法（它接受注册表的 `ResourceKey`，一个包含 `BootstapContext` 的 `RegistryBootstrap` Consumer来注册对象，以及一个可选的 `Lifecycle` 参数来指示注册表的当前生命周期状态）来处理特定注册表进行注册。

```
new RegistrySetBuilder()
  // 创建已配置的特性
  .add(Registries.CONFIGURED_FEATURE, bootstrap -> {
    // 在此处注册已配置的特性
  })
  // 创建已放置的特性
  .add(Registries.PLACED_FEATURE, bootstrap -> {
    // 在此处注册已放置的特性
  });
```

注意

通过Forge创建的数据包注册表也可以通过传递相关的 `ResourceKey` 来使用该生成器生成它们的对象。

使用 BootstapContext 注册

生成器提供的 BootstapContext 中的 #register 方法可用于注册对象。它采用 ResourceKey 表示对象的注册表名称、要注册的对象，以及一个可选的 Lifecycle 参数来指示注册表对象的当前生命周期状态。

```
public static final ResourceKey<ConfiguredFeature<?, ?>> EXAMPLE_CONFIGURED_FEATURE = ResourceKey.create(
  Registries.CONFIGURED_FEATURE,
  new ResourceLocation(MOD_ID, "example_configured_feature")
);

// 在某个恒定的位置或参数中
new RegistrySetBuilder()
  // 创建已配置的特性
  .add(Registries.CONFIGURED_FEATURE, bootstrap -> {
    // 在此处注册已配置的特性
    bootstrap.register(
      // 已配置的特性的资源键
      EXAMPLE_CONFIGURED_FEATURE,
      new ConfiguredFeature<>(
        Feature.ORE, // 创建一个矿物特性
        new OreConfiguration(
          List.of(), // 不做任何事情
          8 // 在最多8个矿脉中
        )
      )
    );
  })
  // 创建已放置的特性
  .add(Registries.PLACED_FEATURE, bootstrap -> {
    // 在此处注册已放置的特性
  });
```

Datapack Registry Object Lookup

有时，数据包注册表对象可能希望使用其他数据包注册表对象或包含数据包注册表对象的标签。在这种情况下，你可以使用 BootstapContext#lookup 查找另一个数据包注册表以获得 HolderGetter 。从那里，你可以通过 #getOrThrow 传递相关的键，获得数据包注册表对象的 Holder$Reference 或标签的 HolderSet$Named 。

```
public static final ResourceKey<ConfiguredFeature<?, ?>> EXAMPLE_CONFIGURED_FEATURE = ResourceKey.create(
  Registries.CONFIGURED_FEATURE,
  new ResourceLocation(MOD_ID, "example_configured_feature")
);

public static final ResourceKey<PlacedFeature> EXAMPLE_PLACED_FEATURE = ResourceKey.create(
  Registries.PLACED_FEATURE,
  new ResourceLocation(MOD_ID, "example_placed_feature")
);

// 在某个恒定的位置或参数中
new RegistrySetBuilder()
  // 创建已配置的特性
  .add(Registries.CONFIGURED_FEATURE, bootstrap -> {
    // 在此处注册已配置的特性
    bootstrap.register(
      // 已配置的特性的资源键
      EXAMPLE_CONFIGURED_FEATURE,
      new ConfiguredFeature(/* ... */)
    );
```

```
})
// 创建已放置的特性
.add(Registries.PLACED_FEATURE, bootstrap -> {
  // 在此处注册已放置的特性

  // 获取已配置的特性的注册表
  HolderGetter<ConfiguredFeature<?, ?>> configured = bootstrap.lookup(Registries.CONFIGURED_FEATURE);

  bootstrap.register(
    // 已放置的特性的资源键
    EXAMPLE_PLACED_FEATURE,
    new PlacedFeature(
      configured.getOrThrow(EXAMPLE_CONFIGURED_FEATURE), // 获取已配置的特性
      List.of() // 并对于放置位置不做任何事情
    )
  )
});
```

15. 杂项功能

15.1 配置

配置定义了可以应用于模组实例的设置和Consumer偏好。Forge使用一个采用TOML文件的配置系统，并使用NightConfig进行读取。

15.1.1 创建一个配置

可以使用 `IConfigSpec` 的子类型创建配置。Forge通过 `ForgeConfigSpec` 实现该类型，并通过 `ForgeConfigSpec$Builder` 实现其构造。生成器可以通过 `Builder#push` 创建一个部分，通过 `Builder#pop` 留下一个部分以将配置值分隔为多个部分。之后，可以使用以下两种方法之一构建配置：

方法	描述
`build`	创建 `ForgeConfigSpec`。
`configure`	创建一对包含配置值和 `ForgeConfigSpec` 的类。

注意

`ForgeConfigSpec$Builder#configure` 通常与 `static` 块和一个类一起使用，该类将 `ForgeConfigSpec$Builder` 作为其构造函数的一部分，用于附加和保存值：

```
// 在某个配置类中
ExampleConfig(ForgeConfigSpec.Builder builder) {
  // 在此处在final字段中定义值
}

// 在该构造函数可被访问的某处
static {
  Pair<ExampleConfig, ForgeConfigSpec> pair = new ForgeConfigSpec.Builder()
    .configure(ExampleConfig::new);
  // 在某个常量字段中存储成对的值
}
```

可以为每个配置值提供额外的上下文，以提供额外的行为。必须先定义上下文，然后才能完全生成配置值：

方法	描述
`comment`	提供配置值的作用说明。可以为多行注释提供多个字符串。
`translation`	为配置值的名称提供翻译键。
`worldRestart`	必须重新启动世界，才能更改配置值。

ConfigValue

配置值可以使用所提供的上下文（如果已定义）使用任何 `#define` 方法构建。

所有配置值方法都至少接受两个组件：

- 表示变量名称的路径：一个被 `.` 分隔的字符串，表示配置值所在的部分
- 不存在有效配置时的默认值

特定于 `ConfigValue` 的方法包含两个附加组件：

- 用于确保反序列化对象有效的验证器
- 表示配置值的数据类型的类

```
// 对于某个ForgeConfigSpec$Builder生成器
ConfigValue<T> value = builder.comment("Comment")
  .define("config_value_name", defaultValue);
```

可以使用 `ConfigValue#get` 获取值本身。这些值会被额外缓存，以防止从文件中进行多次读取。

附加的配置值类型

- 范围值
  - 描述：值必须在所定义的范围之间
  - 类型：`Comparable<T>`
  - 方法名称：`#defineInRange`
  - 附加组件：
  - 配置值可能的最小值和最大值
  - 表示配置值的数据类型的类

> **注意**
>
> `DoubleValue` 、 `IntValue` 和 `LongValue` 是将类型分别指定为 `Double` 、 `Integer` 和 `Long` 的范围值。

- 白名单值
  - 描述: 值必须在所提供的集合中
  - 类型: `T`
  - 方法名称: `#defineInList`
  - 附加组件:
  - 配置所允许的值的集合
- 列表值
  - 描述: 值是一个条目列表
  - 类型: `List<T>`
  - 方法名称: `#defineList` , `#defineListAllowEmpty` （如果列表可为空）
  - 附加组件:
  - 用于确保列表中反序列化元素有效的验证器
- 枚举值
  - 描述: 在所提供的集合中的一个枚举值
  - 类型: `Enum<T>`
  - 方法名称: `#defineEnum`
  - 附加组件:
  - A getter to convert a string or integer into an enum
  - A collection of the allowed values the configuration can be
- 布尔值
  - 描述: A `boolean` value
  - 类型: `Boolean`
  - 方法名称: `#define`

15.1.2 注册一个配置

一旦构建了 `ForgeConfigSpec` ，就必须对其进行注册，以允许Forge根据需要加载、跟踪和同步配置设置。配置应通过 `ModLoadingContext#registerConfig` 在模组构造函数中注册。配置可以注册为表示配置所属侧的给定类型 `ForgeConfigSpec` ，以及配置的特定文件名（可选）。

```
// 在具有一个ForgeConfigSpec CONFIG的模组构造函数中
ModLoadingContext.get().registerConfig(Type.COMMON, CONFIG);
```

以下是可用的配置类型的列表:

类型	被加载	同步到客户端	客户端位置	服务端位置	默认文件后缀
CLIENT	仅在客户端	否	`.minecraft/config`	N/A	`-client`
COMMON	在两端	否	`.minecraft/config`	`<server_folder>/config`	`-common`
SERVER	仅在服务端	是	`.minecraft/saves/<level_name>/serverconfig`	`<server_folder>/world/serverconfig`	`-server`

提示

Forge在相应的代码库中用文档详述了配置类型。

15.1.3 配置事件

每当加载或重新加载配置时发生的操作可以使用 `ModConfigEvent$Loading` 和 `ModConfigEvent$Reloading` 事件来完成。事件必须注册到模组事件总线。

警告

这些事件对于模组的所有配置都被调用；所提供的 `ModConfig` 对象应被用于表示正在加载或重新加载哪个配置。

15.2 键盘布局

键盘布局（键映射）或键盘绑定定义了应与输入绑定的特定操作：鼠标单击、按键等。每当客户端可以进行输入时，都可以检查键盘布局定义的每个操作。此外，每个键盘布局都可以通过控制选项菜单分配给任何输入。

15.2.1 注册一个 KeyMapping

`KeyMapping` 可以通过仅在物理客户端上监听模组事件总线上的 `RegisterKeyMappingsEvent` 并调用 `#register` 来注册。

```
// 在某个仅物理客户端的类中

// 键盘布局是延迟初始化的，因此在注册之前它不存在
public static final Lazy<KeyMapping> EXAMPLE_MAPPING = Lazy.of(() -> /*...*/);

// 事件仅在物理客户端上的模组事件总线上
@SubscribeEvent
public void registerBindings(RegisterKeyMappingsEvent event) {
  event.register(EXAMPLE_MAPPING.get());
}
```

15.2.2 创建一个 KeyMapping

`KeyMapping` 可以使用其构造函数创建。 `KeyMapping` 接受一个定义映射名称的翻译键，映射的默认输入，以及定义映射将放在控制选项菜单中的类别的翻译键。

提示

通过提供原版未提供的类别翻译键，可以将 `KeyMapping` 添加到自定义类别中。自定义类别转换键应包含mod id（例如 `key.categories.examplemod.examplecategory`）。

默认输入

每个键映射都有一个与其关联的默认输入。这是通过 `InputConstants$Key` 提供的。每个输入由一个 `InputConstants$Type` 和一个整数组成，前者定义了提供输入的设备，后者定义了设备上输入的相关标识符。

原版提供三种类型的输入： `KEYSYM` ，通过提供的 `GLFW` 键标记定义键盘， `SCANCODE` ，通过平台特定扫描码定义键盘，以及 `MOUSE` ，定义鼠标。

注意

强烈建议键盘使用 `KEYSYM` 而不是 `SCANCODE` ，因为 `GLFW` 键令牌不与任何特定系统绑定。你可以在GLFW文档上阅读更多内容。

整数取决于提供的类型。所有输入代码都在 `GLFW` 中定义： `KEYSYM` 令牌以 `GLFW_KEY_*` 为前缀，而 `MOUSE` 代码以 `GLFW_MOUSE_*` 作为前缀。

```
new KeyMapping(
  "key.examplemod.example1", // 将使用该翻译键进行本地化
  InputConstants.Type.KEYSYM, // 在键盘上的默认映射
  GLFW.GLFW_KEY_P, // 默认键为P
```

```
  "key.categories.misc" // 映射将在杂项（misc）类别中
)
```

注意

如果键映射不应映射到默认值，则应将输入设置为 `InputConstants#UNKNOWN` 。原版构造函数将要求你通过 `InputConstants$Key#getValue` 提取输入代码，而Forge构造函数可以提供原始输入字段。

IKeyConflictContext

并非所有映射都用于每个上下文。有些映射仅在GUI中使用，而另一些映射仅在游戏中使用。为了避免在不同上下文中使用的同一键的映射相互冲突，可以分配 `IKeyConflictContext` 。

每个冲突上下文包含两种方法： `#isActive` ，定义映射是否可以在当前游戏状态下使用； `#conflicts` ，定义在相同或不同的冲突上下文中映射是否与键冲突。

目前，Forge通过 `KeyConflictContext` 定义了三个基本上下文： `UNIVERSAL` ，这是默认的，意味着密钥可以在每个上下文中使用； `GUI` ，这意味着映射只能在 `Screen` 打开时使用； `IN_GAME` ，意味着映射只有在 `Screen` 未打开时才能使用。可以通过实现 `IKeyConflictContext` 来创建新的冲突上下文。

```
new KeyMapping(
  "key.examplemod.example2",
  KeyConflictContext.GUI, // 映射只能在当一个屏幕打开时使用
  InputConstants.Type.MOUSE, // 在鼠标上的默认映射
  GLFW.GLFW_MOUSE_BUTTON_LEFT, // 在鼠标左键上的默认鼠标输入
  "key.categories.examplemod.examplecategory" // 映射将在新的示例类别中
)
```

KeyModifier

如果修改键保持不变（例如 `G` 与 `CTRL + G` ），则修改器可能不希望映射具有相同的行为。为了解决这个问题，Forge在构造函数中添加了一个额外的参数来接受一个 `KeyModifier` ，它可以将control（ `KeyModifier#CONTROL` ）、shift（ `KeyModifier#SHIFT` ）或alt（ `KeyModifier#ALT` ）映射到任何输入。 `KeyModifier#NONE` 是默认值，不会应用任何修改器。

通过接纳修饰符键和相关输入，可以在控制选项菜单中添加修改器。

```
new KeyMapping(
  "key.examplemod.example3",
  KeyConflictContext.UNIVERSAL,
  KeyModifier.SHIFT, // 默认映射要求shift被按下
  InputConstants.Type.KEYSYM, // 默认映射在键盘上
  GLFW.GLFW_KEY_G, // 默认键为G
  "key.categories.misc"
)
```

15.2.3 检查一个 KeyMapping

可以检查 `KeyMapping` 以查看它是否已被单击。根据时间的不同，可以在条件中使用映射来应用关联的逻辑。

在游戏内

在游戏内，应通过在**Forge**事件总线上监听 `ClientTickEvent` 并在while循环中检查 `KeyMapping#consumeClick` 来检查映射。 `#consumeClick` 仅当输入已执行但之前尚未处理的次数时才会返回 `true` ，因此不会无限拖延游戏。

```
// 事件仅在物理客户端上的Forge事件总线上
public void onClientTick(ClientTickEvent event) {
```

```
  if (event.phase == TickEvent.Phase.END) { // 仅调用代码一次，因为tick事件在每个tick调用两次
    while (EXAMPLE_MAPPING.get().consumeClick()) {
      // 在此处执行单击时的逻辑
    }
  }
}
```

警告

不要将 `InputEvent` 用作 `ClientTickEvent` 的替代项。只有键盘和鼠标输入有单独的事件，所以它们不会处理任何额外的输入。

Inside a GUI

在GUI内，可以使用 `IForgeKeyMapping#isActiveAndMatches` 在其中一个 `GuiEventListener` 方法中检查映射。可以检查的最常见方法是 `#keyPressed` 和 `#mouseClicked` 。

`#keyPressed` 接收 `GLFW` 键令牌、特定于平台的扫描代码和按下的修改器的位字段。通过使用 `InputConstants#getKey` 创建输入，可以根据映射检查键。修改器已经在映射方法本身中进行了检查。

```
// 在某个Screen子类中
@Override
public boolean keyPressed(int key, int scancode, int mods) {
  if (EXAMPLE_MAPPING.get().isActiveAndMatches(InputConstants.getKey(key, scancode))) {
    // 在此处执行按键时的逻辑
    return true;
  }
  return super.keyPressed(x, y, button);
}
```

注意

如果你不拥有要检查键的屏幕，你可以在**Forge**事件总线上监听 `ScreenEvent$KeyPressed` 的 `Pre` 或 `Post` 事件。

`#mouseClicked` 获取鼠标的**x**位置、**y**位置和单击的按钮。通过使用带有 `MOUSE` 输入的 `InputConstants$Type#getOrCreate` 创建输入，可以根据映射检查鼠标按钮。

```
// 在某个Screen子类中
@Override
public boolean mouseClicked(double x, double y, int button) {
  if (EXAMPLE_MAPPING.get().isActiveAndMatches(InputConstants.TYPE.MOUSE.getOrCreate(button))) {
    // 在此处执行鼠标单击时的逻辑
    return true;
  }
  return super.mouseClicked(x, y, button);
}
```

注意

如果你不拥有要检查鼠标的屏幕，你可以在**Forge**事件总线上监听 `ScreenEvent$MouseButtonPressed` 的 `Pre` 或 `Post` 事件。

15.3 游戏测试

游戏测试是运行游戏内单元测试的一种方式。该系统被设计为可扩展的，并可并行高效地运行大量不同的测试。测试对象交互和行为只是该框架众多应用程序中的一小部分。

15.3.1 创建一个游戏测试

一个标准的游戏测试遵循以下三个基本步骤：

1. 加载一个结构或模板，其中包含测试交互或行为的场景（scene）。
2. 一种方法执行要在场景中执行的逻辑。
3. 逻辑执行的方法。如果达到成功状态，则测试成功。否则，测试将失败，结果将存储在场景附近的讲台（lectern）内。

因此，要创建游戏测试，必须有一个现有的模板来保存场景的初始开始状态和一个提供执行逻辑的方法。

测试方法

游戏测试方法是一个 `Consumer<GameTestHelper>` 引用，这意味着它接受一个 `GameTestHelper`，但不返回任何内容。要识别游戏测试方法，它必须具有 `@GameTest` 注释：

```
public class ExampleGameTests {
  @GameTest
  public static void exampleTest(GameTestHelper helper) {
    // 做一些事情
  }
}
```

`@GameTest` 注释还包含配置游戏测试运行方式的成员。

```
// 在某个类中
@GameTest(
  setupTicks = 20L, // 该测试花费20个tick来设置执行
  required = false // 失败将记录到日志，但不会影响批处理的执行
)
public static void exampleConfiguredTest(GameTestHelper helper) {
  // 做一些事情
}
```

相对定位

所有 `GameTestHelper` 方法都使用结构方块的当前位置将结构模板场景中的相对坐标转换为其绝对坐标。为了便于在相对定位和绝对定位之间进行转换，可以分别使用 `GameTestHelper#absolutePos` 和 `GameTestHelper#relativePos`。

结构模板的相对位置可以在游戏中通过test命令加载结构，将玩家放置在所需位置，最后运行 `/test pos` 命令来获得。这将获取玩家相对于玩家200个方块内最近结构的坐标。该命令将相对位置导出为聊天中的可复制文本组件，用作最终的本地变量。

提示

`/test pos` 生成的局部变量可以通过将其附加到命令末尾来指定其引用名称：

```
/test pos <var> # 导出'final BlockPos <var> = new BlockPos(...);'
```

成功完成

游戏测试方法负责一件事：在有效完成时标记测试是否成功。如果在超时之前没有达到成功状态（如 `GameTest#timeoutTicks` 所定义），则测试自动失败。

`GameTestHelper` 中有许多抽象方法，可用于定义成功状态；然而，有四个是非常重要的。

方法	描述
`#succeed`	测试被标记为成功。
`#succeedIf`	如果没有抛出 `GameTestAssertException`，则会立即测试所提供的 `Runnable` 并成功。如果测试在该瞬时tick上没有成功，则将其标记为失败。
`#succeedWhen`	所提供的 `Runnable` 在超时之前每tick都会进行测试，如果对其中一个tick的检查没有引发 `GameTestAssertException`，则会成功。
`#succeedOnTickWhen`	提供的 `Runnable` 在指定的tick上进行测试，如果没有抛出 `GameTestAssertException`，则会成功。如果 `Runnable` 在任何其他tick上成功，则将其标记为失败。

重要

游戏测试每tick都会执行，直到测试被标记为成功。因此，在给定的tick上安排成功的方法必须小心，不要总是在之前的tick上失败。

计划操作

并非所有操作都会在测试开始时发生。操作可以安排在特定的时间或间隔进行：

方法	描述
`#runAtTickTime`	操作将在指定的tick上运行。
`#runAfterDelay`	操作将在当前tick后 `x` tick时运行。
`#onEachTick`	操作在每个tick都会运行。

断言

在游戏测试期间的任何时候，都可以进行断言以检查给定条件是否为真。 `GameTestHelper` 中有许多断言方法；然而，它简化为在不满足适当状态时抛出 `GameTestAssertException`。

生成的测试方法

如果需要动态生成游戏测试方法，则可以创建测试方法生成器。这些方法不接受任何参数，并返回一个 `TestFunction` 的集合。要识别测试方法生成器，它必须具有 `@GameTestGenerator` 注释：

```
public class ExampleGameTests {
  @GameTestGenerator
  public static Collection<TestFunction> exampleTests() {
    // 返回一个TestFunction的集合
  }
}
```

TestFunction

`TestFunction` 是 `@GameTest` 注释和运行测试的方法所包含的包装信息。

> **提示**
>
> 任何使用 `@GameTest` 注释的方法都会使用 `GameTestRegistry#turnMethodIntoTestFunction` 转换为 `TestFunction` 。该方法可以用作创建 `TestFunction` 的引用，而无需使用注释。

批量处理

游戏测试可以批量执行，而不是按注册顺序执行。可以通过提供相同的 `GameTest#batch` 字符串将测试添加到批次中。

批处理本身并没有提供任何有用的东西。但是，批处理可以用于在测试运行的当前存档上执行设置和拆卸（teardown）状态。这是通过用 `@BeforeBatch` 注释方法来完成的，用 `@AfterBatch` 来进行设置或拆卸。 `#batch` 方法必须与提供给游戏测试的字符串匹配。

批处理方法是 `Consumer<ServerLevel>` 引用，这意味着它们接受 `ServerLevel` 而不返回任何内容：

```
public class ExampleGameTests {
  @BeforeBatch(batch = "firstBatch")
  public static void beforeTest(ServerLevel level) {
    // 进行设置（setup）
  }

  @GameTest(batch = "firstBatch")
  public static void exampleTest2(GameTestHelper helper) {
    // 做一些事情
  }
}
```

15.3.2 注册一个游戏测试

游戏测试必须注册后才能在游戏中运行。有两种方法：通过 `@GameTestHolder` 注释或 `RegisterGameTestsEvent` 。这两种注册方法仍然需要用 `@GameTest` 、 `@GameTestGenerator` 、 `@BeforeBatch` 或 `@AfterBatch` 对测试方法进行注释。

GameTestHolder

`@GameTestHolder` 注释注册类型（类、接口、枚举或记录）中的任何测试方法。 `@GameTestHolder` 包含一个具有多种用途的单一方法。在该实例中，提供的 `#value` 必须是模组的mod id；否则，测试将不会在默认配置下运行。

```
@GameTestHolder(MODID)
public class ExampleGameTests {
  // ...
}
```

RegisterGameTestsEvent

`RegisterGameTestsEvent` 也可以使用 `#register` 注册类或方法。事件监听器必须添加到模组事件总线。以这种方式注册的测试方法必须在每个用 `@GameTest` 注释的方法上向 `GameTest#templateNamespace` 提供其mod id。

```
// 在某个类中
public void registerTests(RegisterGameTestsEvent event) {
  event.register(ExampleGameTests.class);
}

// 在ExampleGameTests中
@GameTest(templateNamespace = MODID)
public static void exampleTest3(GameTestHelper helper) {
  // 进行设置（setup）
}
```

> **注意**
>
> 提供给 `GameTestHolder#value` 和 `GameTest#templateNamespace` 的值可能与当前的mod id不同。需要更改buildscript中的配置。

15.3.3 结构模板

游戏测试是在由结构或模板加载的场景中执行的。所有模板都定义了场景的尺寸以及将要加载的初始数据（方块和实体）。模板必须存储为 `data/<namespace>/structures` 中的 `.nbt` 文件。

> **提示**
>
> 可以使用结构方块创建和保存结构模板。

模板的位置由以下几个因素指定：

- 模板的命名空间是否被指定。
- 类是否应被加到模板的名称之前。
- 模板的名称是否被指定。

模板的命名空间由 `GameTest#templateNamespace` 确定，如果未指定则由 `GameTestHolder#value` 确定，如果两者都未指定则由 `minecraft` 确定。

如果将 `@PrefixGameTestTemplate` 应用于具有测试注释的类或方法并设置为 `false` ，则简单类名不会前置于模板的名称。否则，简单类名将变为小写并加上前缀，然后在模板名之前加上一个点。

模板的名称由 `GameTest#template` 决定。如果未指定，则使用方法的小写名称。

```
// 所有结构的modid将为MODID
@GameTestHolder(MODID)
public class ExampleGameTests {

  // 类名已前置，模板名称未指定
  // 模板位置位于'modid:examplegametests.exampletest'
  @GameTest
  public static void exampleTest(GameTestHelper helper) { /*...*/ }

  // 类名未前置，模板名称未指定
  // 模板位置位于'modid:exampletest2'
  @PrefixGameTestTemplate(false)
```

```
  @GameTest
  public static void exampleTest2(GameTestHelper helper) { /*...*/ }

  // 类名已前置，模板名称已指定
  // 模板位置位于'modid:examplegametests.test_template'
  @GameTest(template = "test_template")
  public static void exampleTest3(GameTestHelper helper) { /*...*/ }

  // 类名未前置，模板名称已指定
  // 模板位置位于'modid:test_template2'
  @PrefixGameTestTemplate(false)
  @GameTest(template = "test_template2")
  public static void exampleTest4(GameTestHelper helper) { /*...*/ }
}
```

15.3.4 运行游戏测试

可以使用 `/test` 命令运行游戏测试。 `test` 命令具有高度可配置性；但是，只有少数几个对运行测试很重要：

子命令	描述
`run`	运行指定的测试： `run <test name>`
`runall`	运行所有可用的测试。
`runthis`	运行离玩家15个方块内最近的测试。
`runthese`	运行离玩家200个方块内的测试。
`runfailed`	运行上一次运行中失败的所有测试。

注意

子命令跟在test命令后面： `/test <subcommand>` 。

15.3.5 构建脚本（buildscript）配置

游戏测试在构建脚本（ `build.gradle` 文件）中提供额外的配置设置，以运行并集成到不同的设置中。

启用其他命名空间

如果构建脚本是按照推荐的方式进行设置的，那么只会启用当前mod id下的游戏测试。要使其他命名空间能够从中加载游戏测试，运行配置必须将属性 `forge.enabledGameTestNamespaces` 设置为一个字符串，指定用逗号分隔的每个命名空间。如果属性为空或未设置，则将加载所有命名空间。

```
// 在某个运行配置里面
property 'forge.enabledGameTestNamespaces', 'modid1,modid2,modid3'
```

警告

命名空间之间不能有空格；否则，将无法正确加载命名空间。

游戏测试服务端运行配置

游戏测试服务端是一种运行构建服务端的特殊配置。构建服务端返回所需的失败游戏测试数的退出代码。所有失败的测试都被记录到日志，无论是必需的还是可选的。此服务端可以使用 `gradlew runGameTestServer` 运行。

重要

由于Gradle工作方式的特殊性，默认情况下，如果任务强制退出系统，Gradle守护进程将被终止，导致Gradle运行器报告构建失败。ForgeGradle在默认情况下对运行任务设置强制退出，这样任何子项目都不会按顺序执行。然而，因此，游戏测试服务端总是会失败。

这可以通过使用 `#setForceExit` 方法禁用运行配置上的强制退出来解决：

```
// 游戏测试服务端运行配置
gameTestServer {
    // ...
    setForceExit false
}
```

在其他运行配置中启用游戏测试

默认情况下，只有 `client` 、 `server` 和 `gameTestServer` 运行配置启用了游戏测试。如果另一个运行配置应该运行游戏测试，则 `forge.enableGameTest` 属性必须设置为 `true` 。

```
// 在一个运行配置里面
property 'forge.enableGameTest', 'true'
```

15.4 Forge更新检查器

Forge提供了一个非常轻量级的可选择性加入的更新检查框架。如果任何模组有可用的更新，它将在主菜单和模组列表的'Mods'按钮上显示一个闪烁的图标，以及相应的更改日志。它不会自动下载更新。

15.4.1 入门

你要做的第一件事是在 `mods.toml` 文件中指定 `updateJSONURL` 参数。此参数的值应该是指向更新JSON文件的有效URL。这个文件可以托管在你自己的网络服务器、GitHub或任何你想要的地方，只要你的模组的所有用户都能可靠地访问它。

15.4.2 更新JSON格式

JSON本身有一个相对简单的格式，如下所示：

```
{
  "homepage": "<homepage/download page for your mod>",
  "<mcversion>": {
    "<modversion>": "<changelog for this version>",
    // 列出给定Minecraft版本的所有模组版本，以及它们的更改日志
    // ...
  },
  "promos": {
    "<mcversion>-latest": "<modversion>",
    // 为给定的Minecraft版本声明你的模组的最新"bleeding-edge"版本
    "<mcversion>-recommended": "<modversion>",
    // 为给定的Minecraft版本声明你的模组的最新"stable"版本
    // ...
  }
}
```

这是不言自明的，但需要注意：

- `homepage` 下的链接是当模组过时时将向用户显示的链接。
- Forge使用内部算法来确定你的模组的一个版本字符串是否比另一个"新"。大多数版本控制方案应该是兼容的，但如果你担心方案是否受支持，请参阅 `ComparableVersion` 类。强烈建议遵守Maven版本控制。
- 可以使用 `\n` 将变更日志字符串分隔成多行。有些人更喜欢包含一个简略的变更日志，然后链接到一个提供完整变更列表的外部网站。
- 手动输入数据可能很麻烦。你可以将 `build.gradle` 配置为在构建版本时自动更新此文件，因为Groovy具有本地JSON解析支持。这将留给读者练习。
- 这里可以找到一些例子，例如nocubes、Corail Tombstone和Chisels & Bits 2。

15.4.3 检索更新检查结果

你可以使用 `VersionChecker#getResult(IModInfo)` 检索Forge更新检查器的结果。你可以通过 `ModContainer#getModInfo` 获取你的 `IModInfo` 。你可以在构造函数中使用 `ModLoadingContext.get().getActiveContainer()` 、`ModList.get().getModContainerById(<你的modId>)` 或 `ModList.get().getModContainerByObject(<你的模组实例>)` 来获取 `ModContainer` 。你可

以使用 `ModList.get().getModContainerById(<modId>)` 获取任何其他模组的 `ModContainer` 。返回的对象有一个方法 `#status` ，表示版本检查的状态。

状态	描述
`FAILED`	版本检查器无法连接到提供的URL。
`UP_TO_DATE`	当前版本等于推荐版本。
`AHEAD`	如果没有最新版本，则当前版本比推荐版本更新。
`OUTDATED`	有一个新的推荐版本或最新版本。
`BETA_OUTDATED`	有一个新的最新版本。
`BETA`	当前版本等于或高于最新版本。
`PENDING`	请求的结果尚未完成，因此你应该稍后再试。

返回的对象还将具有 `update.json` 中指定的目标版本和任何变更日志行。

15.5 调试分析器

Minecraft提供了一个调试分析器，它提供系统数据、当前游戏设置、JVM数据、存档数据和tick信息，以查找耗时的代码。考虑到像 `TickEvent` 和计时 `BlockEntities` 这样的事情，这对想要找到滞后源的模组开发者和服务器所有者来说非常有用。

15.5.1 使用调试分析器

调试分析器使用起来非常简单。它需要调试绑定键 `F3 + L` 来启动分析器。10秒后，它将自动停止；但是，可以通过再次按绑定键提前停止。

注意

当然，你只能分析实际到达的代码路径。要分析的实体和方块实体必须存在于存档中才能显示在结果中。

停止调试器后，它将在运行目录中的 `debug/profiling` 子目录中创建一个新的zip。 文件名的格式为日期和时间 `yyyy-mm-dd_hh_mi_ss-WorldName-VersionNumber.zip`

15.5.2 阅读一个分析结果

在每个端位的文件夹（ `client` 和 `server` ）中，你会发现一个包含结果数据的 `profiling.txt` ”文件。在顶部，它首先告诉它运行了多长时间（以毫秒为单位）以及在这段时间内运行了多少tick。

在下面，你将找到与以下片段类似的信息：

```
[00] levels - 96.70%/96.70%
[01] |   Level Name - 99.76%/96.47%
[02] |   |   tick - 99.31%/95.81%
[03] |   |   |   entities - 47.72%/45.72%
[04] |   |   |   |   regular - 98.32%/44.95%
[04] |   |   |   |   blockEntities - 0.90%/0.41%
[05] |   |   |   |   |   unspecified - 64.26%/0.26%
[05] |   |   |   |   |   minecraft:furnace - 33.35%/0.14%
[05] |   |   |   |   |   minecraft:chest - 2.39%/0.01%
```

以下是每个部分的含义的小解释

[02]	tick	99.31%	95.81%
该部分的深度	该部分的名称	所花费的时间相对于其父项的百分比。对于层级0，它是一tick所用时间的百分比。对于层级1，它是其父层所用时间的百分比。	第二个百分比告诉整个tick所花的时间。

15.5.3 分析你自己的代码

调试分析器具有对 `Entity` 和 `BlockEntity` 的基本支持。如果你想分析其他内容，你可能需要手动创建你的部分，如下所示：

```
ProfilerFiller#push(yourSectionName : String);
// 你想分析的代码
ProfilerFiller#pop();
```

你可以从 `Level` 、 `MinecraftServer` 或 `Minecraft` 实例获取 `ProfilerFiller` 实例。 现在，你只需要在结果文件中搜索你的部分的名称。

16. 进阶主题

16.1 访问转换器

访问转换器（简称AT）允许扩大可见性并修改类、方法和字段的 `final` 标志。它们允许模组开发者访问和修改其控制之外的类中不可访问的成员。

规范文档可以在Minecraft Forge GitHub上查看。

16.1.1 添加AT

在你的模组项目中添加一个访问转换器就像在 `build.gradle` 中添加一行一样简单：

```
// 此代码块也是指定映射版本的位置
minecraft {
  accessTransformer = file('src/main/resources/META-INF/accesstransformer.cfg')
}
```

添加或修改访问转换器后，必须刷新Gradle项目才能使转换生效。

在开发过程中，AT文件可以位于上面一行指定的任何位置。然而，当在非开发环境中加载时，Forge只会在JAR文件中搜索 `META-INF/accesstransformer.cfg` 的确切路径。

16.1.2 注释

`#` 之后直到行尾的所有文本都将被视为注释，不会被解析。

16.1.3 访问修饰符

访问修饰符指定给定目标将转换为什么样的新成员可见性。按可见性降序：

- `public` - 对其包内外的所有类可见
- `protected` - 仅对包内和子类中的类可见
- `default` - 仅对包内的类可见
- `private` - 仅对类内部可见

一个特殊的修饰符 `+f` 和 `-f` 可以附加到前面提到的修饰符中，以分别添加或删除 `final` 修饰符，这可以在应用时防止子类化、方法重写或字段修改。

> **警告**
>
> 指令只修改它们直接引用的方法；任何重写方法都不会进行访问转换。建议确保转换后的方法没有限制可见性的未转换重写，这将导致JVM抛出错误（Error）。
>
> 可以安全转换的方法示例有 `private` 方法、`final` 方法（或 `final` 类中的方法）和 `static` 方法。

16.1.4 目标和指令

> **重要**
>
> 在Minecraft类上使用访问转换器时，字段和方法必须使用SRG名称。

类

转换为目标类：

```
<access modifier> <fully qualified class name>
```

内部类是通过将外部类的完全限定名称和内部类的名称与分隔符 `$` 组合来表示的。

字段

转换为目标字段：

```
<access modifier> <fully qualified class name> <field name>
```

方法

目标方法需要一种特殊的语法来表示方法参数和返回类型：

```
<access modifier> <fully qualified class name> <method name>(<parameter types>)<return type>
```

指定类型

也称为“描述符”：有关更多技术细节，请参阅Java虚拟机规范，SE 8，第4.3.2和4.3.3节。

- `B` - `byte`，有符号字节
- `C` - `char`，UTF-16 Unicode字符
- `D` - `double`，双精度浮点值
- `F` - `float`，单精度浮点值
- `I` - `integer`，32位整数
- `J` - `long`，64位整数
- `S` - `short`，有符号short
- `Z` - `boolean`，`true` 或 `false` 值
- `[` - 代表数组的一个维度
- 例如：`[[S` 指 `short[][]`
- `L<class name>;` - 代表一个引用类型
- 例如：`Ljava/lang/String;` 指 `java.lang.String` 引用类型 （注意左斜杠的使用而非句点）
- `(` - 代表方法描述符，应在此处提供参数，如果不存在参数，则不提供任何参数
- 例如：`<method>(I)Z` 指的是一个需要整数参数并返回布尔值的方法
- `V` - 指示方法不返回值，只能在方法描述符的末尾使用
- 例如：`<method>()V` 指的是一个没有参数且不返回任何值的方法

16.1.5 示例

```
# 将Crypt中的ByteArrayToKeyFunction接口转换为public
public net.minecraft.util.Crypt$ByteArrayToKeyFunction

# 将MinecraftServer中的'random'转换为protected并移除final修饰符
protected-f net.minecraft.server.MinecraftServer f_129758_ #random
```

```
# 将Util中的'makeExecutor'方法转换为public,
# 接受一个String并返回一个ExecutorService
public net.minecraft.Util m_137477_(Ljava/lang/String;)Ljava/util/concurrent/ExecutorService; #makeExecutor

# 将UUIDUtil中的'leastMostToIntArray'方法转换为public
# 接受两个long参数并返回一个int[]
public net.minecraft.core.UUIDUtil m_235872_(JJ)[I #leastMostToIntArray
```

17. 向Forge做出贡献

17.1 入门

如果你已经决定为Forge做出贡献，你将不得不采取一些特殊的步骤来开始开发。一个简单的模组开发环境不足以直接使用Forge的代码库。相反，你可以使用以下指南来帮助你进行设置，并开始改进Forge！

17.1.1 fork或克隆（clone）仓库

像你会发现的大多数主要开源项目一样，Forge托管在GitHub上。如果你以前为另一个项目做过贡献，你就会知道这个过程，可以直接跳到下一节。

对于那些通过Git进行协作的初学者来说，以下是两个简单的步骤。

> **注意**
>
> 本指南假设你已经设置了GitHub帐户。如果没有，请访问GitHub的注册页面创建帐户。此外，本指南不是关于git使用的教程。如果你正在努力上手git，请先查询其他资料。

Forking

首先，你必须通过单击右上角的“fork”按钮来“fork”MinecraftForge仓库。如果你在一个组织中，请选择要托管该fork的帐户。

fork仓库是必要的，因为不是每个GitHub用户都可以自由访问每个仓库。相反，你可以创建原始仓库的副本，以便稍后通过所谓的Pull Request贡献你的更改，稍后你将了解更多信息。

Cloning

在fork仓库之后，是时候获得本地访问权限来实际进行一些更改了。为此，你需要将存储库克隆到本地计算机上。

使用你最喜欢的git客户端，只需将你的fork克隆到你选择的目录中。作为一般示例，这里有一个命令行片段，它应该适用于所有正确配置的系统，并将仓库克隆到当前目录下名为“MinecraftForge”的目录中（请注意，你必须将 `<User>` 替换为你的用户名）：

```
git clone https://github.com/<User>/MinecraftForge
```

17.2 检出到正确的分支

fork和克隆存储库是Forge开发的唯一强制性步骤。但是，为了简化为你创建Pull Request的过程，最好使用分支。

建议为你计划提交的每个PR创建并检出一个分支。这样，你就可以在工作于旧补丁的同时，随时了解Forge对新PR的最新更改。

完成此步骤后，你就可以开始设置开发环境了。

17.2.1 设置开发环境

根据你喜欢的IDE，你必须遵循一组不同的推荐步骤才能成功设置开发环境。

Eclipse

由于Eclipse工作区的工作方式，ForgeGradle可以完成大部分相关工作，让你开始使用Forge工作区。

1. 打开终端/命令提示符，然后导航到克隆的fork的目录。
2. 输入 `./gradlew setup` 并按回车。等到ForgeGradle完成。
3. 输入 `./gradlew genEclipseRuns` 并按回车。再次等到ForgeGradle完成。
4. 打开你的Eclipse工作区并转到 `File -> Import -> General -> Existing Gradle Project`.
5. 在打开的对话框中，在“项目根目录”（”Project root directory”）选项中浏览到仓库目录。
6. 点击“完成”按钮完成导入。

这就是让你启动并运行Eclipse所需的全部内容。运行测试模组不需要额外的步骤。只需像在任何其他项目中一样点击“运行”（”Run”），然后选择适当的运行配置。

IntelliJ IDEA

JetBrains的旗舰IDE提供了对Gradle的强大集成支持：Forge的首选构建系统。然而，由于Minecraft模组开发的一些特点，需要额外的步骤才能使一切正常工作。

IDEA 2021及以后版本

1. IntelliJ IDEA 2021，启动！
   - 如果你已经打开了另一个项目，请使用 文件 -> 关闭项目 选项关闭该项目。
2. 在“欢迎使用IntelliJ IDEA”窗口的项目选项卡中，单击右上角的“打开”按钮，然后选择你之前克隆的MinecraftForge文件夹。
3. 如有提示，点击“信任项目”。
4. IDEA导入项目并索引其文件后，运行Gradle设置任务。你可以通过以下方式执行此操作：
   - 打开屏幕右侧的Gradle侧边栏，然后打开forge项目树，选择任务（Tasks），然后选择其他（other），然后双击 forge -> 任务 -> 其他 -> “设置”（ `setup` ） 中的 `setup` 任务（也可能显示为 `MinecraftForge[Setup]` ）。
5. 生成运行配置
   - 打开屏幕右侧的Gradle侧边栏，然后打开forge项目树，选择任务，然后选择其他，双击 forge -> 任务 -> forgegradle runs -> `genIntellijRuns` 中的 `genIntellijRuns` 任务（也可能显示为 `MinecraftForge[genIntellijRuns]` ）。
   - 如果在进行任何更改之前在构建过程中遇到许可错误，运行 `updateLicenses` 任务可能会有所帮助。这个任务也可以在 Forge -> Tasks -> other 中找到。

IDEA 2019-2020

IDEA 2021和这些版本的设置之间有一些细微的差异。

1. 导入Forge的 `build.gradle` 作为IDEA项目。为此，只需从 `Welcome to IntelliJ IDEA` 启动屏幕中单击 `Import Project` ，然后选择 `build.gradle` 文件。
2. IDEA导入项目并索引文件后，运行Gradle设置任务。两者之一：
   1. 打开屏幕右侧的Gradle侧边栏，然后打开 `forge` 项目树，选择 `Tasks` ，然后选择 `other` ，双击 `setup` 任务（也可能显示为 `MinecraftForge[Setup]` ）。或者：
   2. 按CTRL键两次，然后在弹出的 `Run` 命令窗口中键入 `gradle setup` 。

然后，你可以使用 `forge client` Gradle任务（ `Tasks -> fg runs -> forge client` ）运行Forge：右键单击任务并根据需要选择 `Run` 或 `Debug` 。

你现在应该能够使用你对Forge和原版代码库所做的更改来使用你的模组。

17.2.2 做出更改并提交Pull Request

一旦你设置了你的开发环境，是时候对Forge的代码库进行一些更改了。然而，在编辑项目代码时，你必须避免一些陷阱。

最重要的是，如果你想编辑Minecraft的源代码，你必须只在“Forge”子项目中这样做。“Clean”项目中的任何更改都会干扰ForgeGradle和生成补丁。这可能会带来灾难性的后果，并可能使你的环境完全无用。如果你希望拥有完美的体验，请确保你只在“Forge”项目中编辑代码！

生成补丁

在你对代码库进行了更改并对其进行了彻底测试之后，你可以继续生成补丁。只有当你在Minecraft代码库中工作时（即在“Forge”项目中），这才是必要的，但这一步骤对于你的更改在其他地方工作至关重要。Forge的工作原理是只将更改后的内容注入原版Minecraft，因此需要以适当的格式提供这些更改。值得庆幸的是，ForgeGradle能够生成变更集供你提交。

要启动补丁生成，只需从IDE或命令行运行 `genPatches` Gradle任务。完成后，你可以提交所有更改（确保没有添加任何不必要的文件）并提交Pull Request！

Pull Requests

将你的贡献添加到Forge之前的最后一步是Pull Request（简称PR）。这是一个将fork的更改合并到活动代码库中的正式请求。创建PR很容易。只需转到这个GitHub页面并按照建议的步骤进行操作。现在，对于分支的良好设置是有回报的，因为你可以准确地选择要提交的更改。

> **注意**
>
> Pull Request受规则约束；并不是每一个请求都会被盲目接受。关注本文档以获取更多信息并确保你的PR达到最佳质量！如果你想最大限度地提高你的PR被接受的机会，请遵循这些PR准则！

17.3 Pull Request准则

模组是在Forge之上构建的，但有些事情Forge不支持，这限制了模组的功能。 当模组开发者遇到类似的情况时，他们可以对Forge进行更改以支持它，并将该更改作为Pull Request提交到Github上。

为了充分利用你和Forge团队的时间，建议在准备Pull Request时遵循一些粗略的指导原则。在编写一个好的Pull Request时，以下几点是需要记住的最重要的方面。

17.3.1 到底什么是Forge?

在较高级别上，Forge是Minecraft之上的一个模组兼容性层。 早期的模组直接编辑了Minecraft的代码（就像现在的coremod一样），但当他们编辑相同的东西时，他们会遇到冲突。当一个模组以其他模组无法预料的方式改变行为时（就像现在的coremod一样），他们也遇到了问题，导致了神秘的问题和很多头痛。

通过使用Forge之类的东西，模组可以集中常见的更改并避免冲突。 Forge还包括通用模组功能的支持结构，如Capability、注册表和其他允许模组更好地协同工作的功能。

在编写一个好的Forge Pull Request时，你还必须知道Forge在较低级别上是什么。 Forge中有两种主要类型的代码：Minecraft补丁和Forge代码。

17.3.2 补丁

补丁是作为对Minecraft源代码的直接更改应用的，且致力于尽可能最小化。 每次Minecraft代码更改时，都需要仔细查看所有Forge补丁，并将其正确应用于新代码。 这意味着，改变很多事情的大型补丁很难维护，因此Forge的目标是避免这些补丁，并使补丁尽可能小。 除了确保代码有意义之外，对补丁的审查将侧重于最小化大小。

制作小补丁有很多策略，评论通常会指出更好的方法。 Forge补丁程序通常插入一行触发事件或代码挂钩，如果事件满足某些条件，就会影响之后的代码。 这允许大多数代码存在于补丁之外，从而使补丁保持小而简单。

有关创建补丁的更多详细信息，请参阅GitHub wiki。

17.3.3 Forge代码

除了补丁之外，Forge代码只是普通的Java代码。它可以是事件代码、兼容性功能，也可以是任何不直接编辑Minecraft代码的东西。 当Minecraft更新时，Forge代码必须像其他一切一样更新。然而，它要容易得多，因为它没有直接纠缠在Minecraft代码中。

因为这个代码是独立的，所以没有像补丁那样的大小限制。

除了确保代码有意义之外，评审（review）还将侧重于使代码干净：使用正确的格式和Java文档。

17.3.4 解释你自己

所有Pull Request都需要回答一个问题：为什么这是必要的? 添加到Forge中的任何代码都需要维护，而更多的代码意味着更可能出现错误，因此添加代码需要有充分的理由。

一个常见的Pull Request问题是没有提供解释，或者给出了理论上如何使用Pull Request的神秘例子。 这只会延迟Pull Request过程。 对一般情况有一个明确的解释是好的，但也给出了一个具体的例子，说明你的模组如何需要这个Pull Request。

有时有更好的方法来做你想做的事情，或者一种完全不需要Pull Request的方法。在完全排除这些可能性之前，代码更改不能被接受。

17.3.5 证明它有效

你提交给Forge的代码应该能完美地工作，这取决于你是否能说服评审人员。

最好的方法之一是在Forge中添加一个示例模组或JUnit测试，利用你的新代码并展示它的工作原理。

要使用示例模组设置和运行Forge环境，请参阅这个指南。

17.3.6 Forge中的突破性变化

Forge不能做出破坏依赖它的模组的更改。 这意味着Pull Request必须确保它们不会破坏与以前Forge版本的二进制兼容性。 破坏二进制兼容性的更改称为“突破性变化”。

关于这个有一些例外：

- Forge在新的Minecraft版本开始时接受突破性变化，因为Minecraft本身已经为模组开发者造成了突破性变化。
- 有时需要在该时间窗口之外进行紧急更改，但这种情况很少见，可能会给改装后的Minecraft社区中的每个人带来依赖性头痛。

在这些特殊时间之外，不接受具有突破性变化的Pull Request。它们必须适应以支持旧的行为，或者等待下一个Minecraft版本。

17.3.7 有耐心、有礼貌、有同情心

在提交Pull Request时，你通常必须通过代码审查并进行多次更改，才能获得最佳的Pull Request。 请记住，代码审查并不是针对你的判断。代码中的错误不是针对个人的。没有人是完美的，这就是我们合作的原因。

消极也无济于事。威胁放弃你的Pull Request，转而编写一个coremod，只会让人们感到不安，并使修改后的生态系统变得更糟。 重要的是，在一起工作时，你要考虑到审查你的Pull Request的人的最佳意图，不要把事情看得太个人化。

17.3.8 审查（Review）

如果你尽最大努力理解Pull Request流程的缓慢和完美主义本质，我们也会尽最大努力了解你的观点。

在你的Pull Request经过审查并尽所有人所能进行清理后，它将被Lex标记为最终审查，Lex对项目中包含的内容拥有最终发言权。

18. 旧版本

18.1 旧版本的文档

Forge已经存在多年了，你仍然可以轻松访问Minecraft 1.1版本的Forge版本。每个版本之间都有显著的差异，支持这么多不同的版本是不可能的。因此，Forge使用了一个LTS系统，其中以前的主要Minecraft版本被视为“LTS”（长期支持）。只有最新版本和任何当前的LTS版本才会有易于访问的文档，并包含在侧边栏的版本下拉列表中。然而，一些旧版本曾经是LTS，或者在某个时候是最新版本，并编写了文档。可以在这里找到带有这些版本文档的旧网站链接。

重要

这些旧文档网站仅供参考。不要在Forge discord或Forge论坛上寻求旧版本的帮助。当你使用旧版本时，将不会获得支持。

以前已有文档的版本列表

不幸的是，并非所有版本都使用了相当长的时间，那些版本的文档可能不完整。无论何时发布新版本，都会随着时间的推移复制和调整上一版本的文档，以包含新的和更新的信息。当某个版本长期不受支持时，信息永远不会更新。准确率百分比表示本应更新的信息实际更新了多少。

版本	准确率	链接
1.12.x	100%	https://docs.minecraftforge.net/en/1.12.x/
1.13.x	10%	https://docs.minecraftforge.net/en/1.13.x/
1.14.x	10%	https://docs.minecraftforge.net/en/1.14.x/
1.15.x	85%	https://docs.minecraftforge.net/en/1.15.x/
1.16.x	85%	https://docs.minecraftforge.net/en/1.16.x/
1.17.x	85%	https://docs.minecraftforge.net/en/1.17.x/
1.18.x	90%	https://docs.minecraftforge.net/en/1.18.x/
1.19.2	90%	https://docs.minecraftforge.net/en/1.19.2/

RetroGradle

RetroGradle是一项档案计划，旨在更新旧版ForgeGradle 1.x至2.3工具链及其Minecraft版本，以使用现代ForgeGradle 4.x及以上工具链。目标是通过将Minecraft Forge的所有过去发布的版本移动到可验证的工作和现代工具链来保留它们，该工具链是数据驱动的，而不是针对特定版本的工作流进行硬编码。

如果任何开发人员希望为这项档案工作做出贡献，请访问The Forge Project discord服务器，并询问指定频道的方向。请注意，此计划仅旨在为社区的利益 保留 这些旧版本， 而不是支持为这些不受支持的旧版本开发模组。 不支持使用或开发不受支持的版本。

18.2 移植到Minecraft 1.19

在这里，你可以找到如何从旧版本移植到当前版本的入门资料列表。有些版本被集中在一起，因为那个特定的版本从未有过太多的用途。

从 -> 到	入门资料
1.12 -> 1.13/1.14	Primer by williewillus
1.14 -> 1.15	Primer by williewillus
1.15 -> 1.16	Primer by 50ap5ud5
1.16 -> 1.17	Primer by 50ap5ud5
1.19.2 -> 1.19.3	Primer by ChampionAsh5357
1.19.3 -> 1.19.4	Primer by ChampionAsh5357